University of Alberta

LEGACY INTERFACE MIGRATION:
– FROM GENERIC ASCII UIS TO TASK-CENTERED GUIS

by

Lanyan Kong

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-60135-8

Canada

University of Alberta

Library Release Form

Name of Author: Lanyan Kong

Title of Thesis: Legacy Interface Migration: – From Generic ASCII UIs to Task-Centered GUIs

Degree: Master of Science

Year this Degree Granted: 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Lanyan Kong

Lanyan Kong
615 General Services Building
University of Alberta
Edmonton, Alberta
Canada, T6G 2H1

Date: April 17, 2000.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Legacy Interface Migration: – From Generic ASCII UIs to Task-Centered GUIs** submitted by Lanyan Kong in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Eleni Stroulia
Supervisor

Dr. Paul G. Sorenson

Dr. Francis Lau
External Examiner

Date: April 16, 2000.

Abstract

Legacy systems represent a considerable investment for many organizations, and hold valuable corporate data. However, they are often difficult to learn and to use because of their ASCII interfaces. A potential solution is to construct a model of how the legacy interface works, and based on it, to develop a Graphical User Interface (GUI) as a front-end. Our approach to developing such a solution is implemented in a prototype system, i.e. URGenT (User interface Re-GENeration Tool). URGenT assists with the migration from the generic ASCII interfaces to the task-centered GUIs.

To understand how the legacy interface works, URGenT analyzes the task interaction, which is recorded as "traces" while users actually perform their regular tasks with the legacy interface, thus avoiding tedious code study or the lengthy interviews with the system users. Using this approach, URGenT recovers the task semantics and a navigation plan to achieve this task, and describes them in a well-defined representation, i.e. a model. Based on the model and a set of GUI-design guidelines, URGenT generates a conceptual design of the task-centered GUI. The increased user-friendliness of GUIs as compared to ASCII interfaces, and the increased efficiency of task-centered interfaces as opposed to generic interfaces will hopefully help to prolong the usage of legacy systems.

Acknowledgements

I would like to express my thanks to Dr. Eleni Stroulia, my supervisor, for her invaluable guidance and support. It is impossible to finish the thesis without her keeping me focused and motivated and her insightful comments. Dr. Stroulia played an important role in the formulation of the thesis topic. She led me into the idea of the thesis topic, and shaped the research continuously with her sharp and brilliant thought. She has taken tremendous time in giving guidance for the thesis writing.

I am also grateful for the suggestion and help from team members in CELLEST project. Special thanks to Paul Sorenson, Roland Penner, Bruce Matichuk and Mohammad El-Ramly. Their inspiring discussion always brings new sparkle into the project, and helps it become more and more mature.

Finally I would like to thank my parents for giving me such a wonderful life and the attitude of appreciation toward the life. My life could never be so happy and promising without my parents' and sister's unconditional love and support. Together with them, everything seems possible!

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Background . . . 2
  - 1.2.1 Reverse Engineering . . . 2
  - 1.2.2 Forward Engineering . . . 4
- 1.3 Thesis Objectives . . . 5
- 1.4 Methodology . . . 6
- 1.5 Anticipated Contributions . . . 7
- 1.6 Organization . . . 8

2 System Architecture 10
- 2.1 Overview of **URGenT** Architecture . . . 12
- 2.2 MiddleWare Introduction . . . 13
- 2.3 **URGenT** and MiddleWare Interactions . . . 14
- 2.4 A Generic Interface Re-Engineering Architecture . . . 15
- 2.5 Evaluation Criteria for Thesis Work . . . 18
  - 2.5.1 Criteria for GUIs of **URGenT** . . . 18
  - 2.5.2 Criteria for **URGenT** . . . 19

3 Task and Domain Modeling 21
- 3.1 Traces Understanding . . . 22
  - 3.1.1 Recorded Traces . . . 23
  - 3.1.2 Initial Task Model . . . 27
- 3.2 Semantics Modeling . . . 29
  - 3.2.1 "Information Exchange" Analysis . . . 29
  - 3.2.2 Analysis Validation . . . 33
  - 3.2.3 Task Modeling . . . 36
- 3.3 Domain Modeling . . . 40
  - 3.3.1 Constant Objects . . . 40
  - 3.3.2 Semantics for Objects . . . 42

4 GUI Development 44
- 4.1 Conceptual Interface Specification . . . 45
  - 4.1.1 External Objects . . . 46
  - 4.1.2 Internal Objects . . . 49
- 4.2 Graphical Interface Generation . . . 50
  - 4.2.1 Graphical Objects Selection . . . 51
  - 4.2.2 Interface Layout . . . 53
  - 4.2.3 Model-based GUI Design Issues . . . 54
  - 4.2.4 Run-time Execution . . . 56

5 Multiple System Interaction 59
- 5.1 Multiple System Recording . . . 60
- 5.2 Interaction Annotation . . . 62
- 5.3 Run-time Execution . . . 63

6 Related Work **65**
6.1 Program Understanding 66
6.2 Model-based Interface Design 67
6.2.1 Adept 69
6.2.2 TRIDENT 69
6.2.3 Mecano 70
6.2.4 MasterMind 71

7 Conclusions And Future Work **73**
7.1 Evaluation 74
7.2 Limitations and Future Work 76
7.2.1 Non-Deterministic Task Analysis 77
7.2.2 User-Support Tools 77
7.2.3 Advanced GUI Design 78
7.3 Contributions 79

Bibliography **82**

List of Tables

3.1 One Task-Specific Trace Recorded in **Recorder** 25
3.2 The Classification Result from Analysis in **URGenT** 32
3.3 Knowledge of Constants 41
3.4 Knowledge of Variables Semantics 42

4.1 Knowledge of Variables Values 55
4.2 Knowledge of Objects Selection 56

5.1 Extended Function Recorded in **URGenT** 61
5.2 Change of The Analyzed Trace After Functional Extension in **URGenT** . . . 62

List of Figures

1.1 Generic Framework for Interface Generation. 4

2.1 Framework of Interface Migration in **URGenT**. 12
2.2 Relations Between the legacy system, The CelWare Toolkit, And **URGenT**. 15
2.3 Generic Framework for Interface Modeling. 16

3.1 Task "Claim Report" in The Legacy Interface. 24
3.2 Relationship of Elements in Initial Task Model. 27
3.3 Task Viewer for Information Flow of "claim report". 34
3.4 Relationship of Elements in Semantic Task Model. 37
3.5 Several Data Specification in *Semantic Task Model* for "claim report" 39

4.1 Conceptual Interface Specification for Task "claim report" 47
4.2 One Screen in The Developed GUI for "claim report" 52

5.1 Specification of Extended Interactions 64

Chapter 1

Introduction

1.1 Motivation

The objective of this thesis is to develop a technique for the *Legacy Interface Re-engineering*, which henceforth is called *Interface Migration*, based on traces of the user's interaction with the legacy system. The thesis research results in the generation of models and methods that assist *Interface Migration*, and implements a prototype system to evaluate those methods.

Legacy systems are the software systems, which were developed using the technology of the 1970s to mid-1980s, and have been deployed by institutions or companies for a long period of time [39]. They implement diverse business policies and decisions, hold valuable corporate data, represent considerable investment from organizations and have years of proven reliability and efficiency. Therefore, they constitute one of the most important assets of the organizations that own them. Unfortunately, there is usually little supporting documentation for these systems, or even none at all, which makes the training of new personnel on how to use them quite difficult. To make matters even worse, most of these systems have text-based interfaces which offer very limited ways to organize and present information to the user, and are therefore usually non-intuitive and difficult to learn and to use [21]. In order to reduce the cost of training users and prolong the usage of legacy systems, it is crucial to make easy-to-use graphical interfaces as front-ends of these systems.

Nowadays, re-engineering legacy systems to prolong their usage is an important area of the software engineering research [26]. *Re-engineering* is the process of restructuring, redesigning, and reimplementing the existing system in a new form, in order to improve maintainability and extend functionality of the existing system. The rationale for re-engineering legacy systems instead of developing new software from scratch is based not only on economic but also pragmatic considerations.

- First, the development of the software is the most difficult and costly component in the computer world [7].
- Secondly, a legacy system is very often the sole repository of valuable corporate knowl-

edge collected over a long time and of the logic within the organization's business processes [21].

- Finally, corporations simply cannot risk disturbance of their business operation by using some new software that would operate their business differently [39].

Therefore it is very important that the software retains most of its properties and functionality. As a result, *Interface Migration* emerges as a very promising solution to the problems described in the first paragraph.

1.2 Background

This section briefly introduces the state of the art in the area of *Re-engineering* and *Interface Migration*. Numerous methods are developed in current research, such as program understanding, data analysis, and knowledge-based components transformation [26]. In general, *Re-engineering* consists of two major processes, *reverse engineering* and *forward engineering*.

- *Reverse engineering* [26] is concerned with the extraction of the functional requirements and the process logic of the legacy system by system analysis and plan recognition, and the representation of the captured information in abstract model or understandable language to support maintenance and evolution.
- *Forward engineering* is the process of generating code from a requirements specification that integrates new technology with the original design of the system.

1.2.1 Reverse Engineering

Reverse engineering involves the process of

- capturing the functional requirements and logic of the legacy system,
- identifying the components of the legacy system and their interrelationships and
- creating representations of the legacy system in another form or a higher level of abstraction.

Possible information sources for a legacy system include its code, its existing design documentation (if available), personal experience and general knowledge about its application domains, or a combination of all the above. The remaining part of the section introduces two methods of obtaining information of the legacy system from these sources.

Program Understanding

Since in general, the source code seems to be a precise and direct definition of the system, a lot of research in *Reverse Engineering* use *program understanding* as the method of collecting knowledge about the legacy system. **Synchronized Refinement** [35, 11], a domain-based program understanding method, is introduced here as an example of Program Understanding based on the system code and on the analyst's view of the system.

Synchronized Refinement first generates two models, i.e. *code model* and *application-domain model*.

(1) The *Application-domain model* is the high-level guide for program understanding. It is derived from existing textual descriptions of the domain by *word-frequency analysis* and a *noun and verb analysis* [31].

(2) *Code model* is the low-level description of the program. It starts as a rough architectural description of the program with some structural and behavioral information from the source code, and is derived by three primary processes: *invocation analysis*, *type analysis* and *coupling analysis*.

The analysis processes involved in the modeling are defined as follows.

- The *invocation analysis* process constructs an initial architectural model by determining which subprograms invoke others in a program. It is useful for generating abstract program understanding even though it is not efficient with all the programming languages.
- The *type analysis* process develops an object-oriented representation of the code, such as **Part-of** associations, by analyzing source code data types.
- The *coupling analysis* process measures the strength of the association between code modules by the amount of communication between them.

After the generation of the two models, **Synchronized Refinement** matches the *code model* to expectations of possible solutions derived from the *application-domain model*, and incrementally transforms and refines both models. *Code model* becomes more abstract by suppressing operational details, while *application-domain model* becomes more specific by augmenting the problem description with solution details. Finally, the two models match "in the middle" and form a complete description of the system from problem statement to implementation.

Although program understanding seems to be able to get a precise description of the legacy system, this is not always possible with the real-world systems.

- First, it assumes that the program is well structured. However, for large systems, because of the complexity and sheer volume of the code and the requisite knowledge

of both source and target environments, along with the possibility of bad structure for the code, such method may be ineffective [39].

- Furthermore, it's not unusual that legacy systems have been modified many times by different programmers, and their original design has been compromised with layers of modifications by people not involved in their original design. As a result, it becomes extremely complex and difficult to elicit consistent and complete understanding for these legacy systems from the "glue" code.

User Interviews

To understand the information and the process logic the legacy system embodies, another approach is to attempt to capture the system's design (data structures and control of processing) by having direct interviews with the expert user [8]. This method is especially useful in some cases that the code is only available in its executable form. However, since legacy systems tend to be large and complex, it is unlikely that one single person in a corporation can completely describe what was in the system [39], and time-consuming iteration is almost the fate of direct *user interviews*. As a result, collecting information about the legacy system using this method can be prohibitively costly and inefficient.

1.2.2 Forward Engineering

Forward Engineering is the process of moving from high-level abstractions and logical, implementation-independent designs of the system to the system physical implementation level [26]. A lot of research has been devoted to the automation of this process, and has produced technologies in different aspects to facilitate the automation. In order to generate the user interface that really works with the system, the process needs not only the static view of the interface, but also the description of its dynamic behavior.

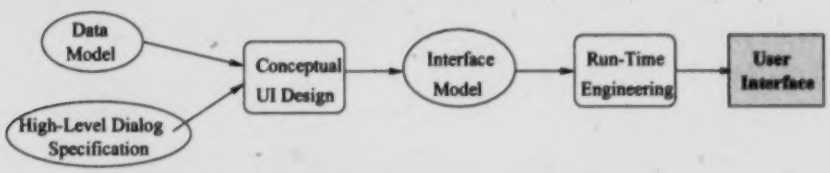

Figure 1.1: Generic Framework for Interface Generation.

Since in systems with GUIs, nearly 50% of source code is devoted to GUI generation [32], it becomes extremely important in the *forward engineering* process to automate the construction of GUI. Currently, there are many tools, such as ITS, UIDE and HUMANOID [24], on the market to support building the user interfaces automatically. They share a generic framework for automating interface-generation from the data model [4], as depicted in Figure 1.1. This framework first produces an interface design by integrating a data

model with a dialog specification, then implements the design using run-time system. They can generate the static layout of the interface from the application data model by using an intelligent tool that applies design rules. However, they all require the specification of the dynamic behavior of the interface separately from the static interface view. For example, UIDE [4](User Interface Development Environment) generates static layout from an extended data model by using a tool, but specifies the dynamic behavior by separately defining sets of pre- and post-conditions for each one of the interface objects.

An alternative approach is to replace the data model in the framework with a domain model. A domain model is a high-level knowledge representation, which captures all the definitions and relationships of the system domain. By using a domain model as input, an intelligent designer can develop both the dynamic dialog specification and the static layout of the interface, and stores them in an interface model, which contains all facets of an interface design including interface objects, presentation, dialog and behavior. Since this interface model can be transformed into UIMS specification, it facilitates the generation of the interface, which can be implemented by a run-time system. Therefore, using this method, both the static layout and the dynamic behavior of the interface can be generated directly from models. One project – **Mecano** [4] that uses this method – is introduced in the chapter six, "Related works".

1.3 Thesis Objectives

Traditional re-engineering strategies focus on evolving the functional architecture of the system, which involves redesigning the user interface and database transactions, as well as converting the language and environment features so that the system could be supported on numerous platforms and provide faster access to application data.

With research in the area growing, it is believed that in order to prolong the usage of the legacy system, the user interface of the legacy system can be reconstructed and made more appropriate for current and upcoming requirements, rather than the *Re-engineering* of the whole system [26]. This thesis investigates a method for interaction-based *Reverse Engineering* for *Interface Migration*. Two problems with the *Legacy Interface* are addressed in the thesis.

- The first problem is the non-intuitive feature of the text-based user interfaces. Our solution to make system easy-to-use is to use Graphical User Interfaces (GUIs) as the front-ends of the legacy systems, because in general, GUIs are easier to learn and to use, as well as capable of promoting users' productivity [5]. In fact, making legacy systems more usable by developing for them graphical interfaces has emerged as one of the most interesting and challenging research in the software industry.

- The second problem is the generic interface design of the legacy system, which builds the user interface initially for the requirements of original classes of tasks in the system domain. The legacy system has evolved during its implementation, merging with other systems, adding functions by glue code, and so on. Moreover, users with different tasks may use different screens and transitions of the existing system interfaces, input and retrieve different types of information, and generally, they have different views of the underlying system. Therefore, the existing interface may or may not meet the needs of all kinds of new tasks and users. Our solution is the task-centered user interface design, which observes the "tasks" as they are now performed and develops one interface to support each particular task. This scheme also makes it possible to optimize the user-system interaction according to each task. A legacy system with such interface may attract many clients, because corporations often want to develop different GUIs for each of their departments to use existing system separately.

1.4 Methodology

In order to accomplish the goal of *Interface Migration*, which enhances the understandability and usability of the legacy system, a three-phase process is developed in this thesis.

- Capture of The *Interaction Traces*.
  Rather than using code studying or direct user interview, this thesis recaptures the system design based on how users accomplish their tasks by their interactions with the system. The interface requirements are extracted by studying all the actions of the user, and the response from the system interface, which are recorded when the user actually performs a task with the system.

- Specification of The *Semantic Task Model*.
  This phase adopts an "information exchange" plan for analyzing the interaction traces, which are recorded in the first phase. The *Semantic Task Model* is a high-level abstraction, which represents the semantics of the task, not just the interactive operation on the user interface. Therefore, it can be mapped into the *Conceptual Interface Model*, which defines the functional requirements for the target interface, independent of the target platform. By integrating the interface functional requirements with new functionality, the *Conceptual Interface Model* is capable of supporting the development of the target interface, which optimizes the system-user interaction as well as enhances the system functionality.

- Generation of The Graphical User Interface.
  The *Conceptual Interface Model* generated in the second level doesn't decide the look and feel of the target interface, therefore doesn't restrict the imagination and creativity

of the designer. In this level, *Conceptual Interface Model* integrates with guidelines on how to organize the new GUI, i.e. what screens with what components are needed in the interface, the functionality and the property of each component, the interrelation among components, and the transition (i.e. interaction process) among screens, finally generates the graphical interface.

1.5 Anticipated Contributions

The overall contribution of this thesis is the development of a method for designing and developing GUIs on front-ends to legacy systems. The concrete work of the project is to develop a prototype system to support the design of task-centered GUIs as the front-ends of the legacy systems. The prototype system developed by the thesis project is **URGenT**, i.e. User interface Re-GENeration Tools. **URGenT** implements a process for inferring the semantics of a specified task and for automating the GUI generation for the task, so that the method developed by the project can be evaluated by real experiments. This method has been implemented and evaluated in **URGenT**. More specifically this thesis makes contributions to the following research problems.

- *Dynamic Behavior Understanding*:
  URGenT understands the task functionality of the user interface by studying the interaction traces, which are recorded when users are actually performing the task with the system.

- *Semantic Task Modeling*:
  URGenT specifies the task objects after classifying their interaction type, i.e. user input or system response, and information scope, i.e. with fixed value in the system or with value that is local to each user-system interaction.

- *Domain Modeling*:
  URGenT collects domain knowledge, i.e. information objects in the system and their attributes, and represents it in a model.

- *GUI Implementation*:
  It generates a new GUI which integrates with the existing system, without any modification of the original system code, and optimizes the performance of the task.

Let us now illustrate **URGenT**'s approach to these issues with an example. Suppose there is a lawyer who wants to develop a report in an insurance legacy system, which contains clients' personal information, accident data and expense summary of accident insurance. This example is called "claim report" task, and will be used all through the thesis to describe the processing of **URGenT**.

First, **URGenT** uses as input the "traces" stored by the terminal emulator which has all the data but is never intended to be used by lawyers for such tasks. The user interacts with the terminal emulator, which asks the user to perform several times of the same task, and records the interactions when the user is actually performing the task. As for this example, the user develops several times the same kind of report for different clients with the legacy interface. Meanwhile, the middleware records the data the user enters, the screen she navigates to, the function she selects, and the information she retrieves. Each performance of the report task is recorded in one "trace".

Then, **URGenT** inspects the traces and recovers the semantic of the task by classifying the information flow in the task. After analyzing multiple recorded traces for the task, **URGenT** concludes the task functionality with different data semantics and types for this task. It uses several steps to accomplish the data classification. The detail is introduced in the chapter three, "Semantic Task and Domain Modeling". **URGenT** adopts a three-tier architecture, which includes the legacy system, the middleware and the **URGenT** system as the three levels. This architecture enables **URGenT** to retrieve information from the legacy system by taking advantage of the middleware, and the developed GUI to manipulate the legacy system through the middleware that supports communication between GUIs and the text-based interfaces of the legacy system.

Finally, from the *Semantic Task Model* which represents the task semantics by the classification of its information flow, **URGenT** generates the *Conceptual Interface Model*, which is independent of the platform the target interface will build on, to describe the functionality that must be embraced by the target interface in order to support the accomplishment of this task. Now, according to one particular user type, for example, the expert user or the novice user, **URGenT** integrates the *Conceptual Interface Model* with the associated interface design guidelines, and generates a GUI for supporting the user to develop the report. The detail is introduced in the chapter four, "GUI Development".

1.6 Organization

The rest of the thesis is organized as follows: Chapter 2 introduces the architecture for the system developed in the thesis research, together with several terminology used in the thesis, and presents the differences between ours and the traditional architecture of re-engineering process. Chapter 3 illustrates the process of the task analysis, the structure of the task model, which contains the components of interface interaction for the task and the relationship between those components, and the *Domain Modeling* in this process. Chapter 4 discusses the content for the GUI design guideline, and elaborates on the method of automating GUI generation and how to support the *Domain Modeling*. Chapter 5 covers the methods of extending the interface functionality by integrating interactions, and the strate-

gies of supporting user participation in the analysis and interface design process. Chapter 6 introduces the related work of *Interface Migration*, and compares their methods with ours. Chapter 7 summarizes and evaluates the contribution of our system and the promising future work in this area.

Chapter 2

System Architecture

In order to pursue the goal of assisting with the *Interface Migration* of the legacy system, which is composed of *Reverse Engineering* and *Forward Engineering*, this thesis addresses two subproblems.

- The first subproblem is developing an *Interface Modeling* method, which infers the task requirements of the legacy interface, represents them as the task semantics in well-defined model, as well as integrates this existing task functionality with other functionality from other applications, for example, the standard desktop tools.
- The second subproblem is developing an *Interface Development* method, which supports the automated model-based generation of interface, and enables the user to perform the task with the system in the newly developed interface that optimizes the task performance.

The two subproblems are achieved by **URGenT**'s two primary processes, *Semantic Task Modeling*, and *GUI Development*, respectively.

- *Semantic Task Modeling*

  URGenT is abductive in nature because it constructs specification of tasks from examples. The first process is analyzing and discovering the functional requirements of a task. In order to do that, this process is composed of the following steps:

  (a) First, it takes as input the recorded traces that capture performance between the expert user and the *Legacy Interface* of the same task.

  (b) Then, it extracts the task semantics by classifying the information flow between users and the legacy system, as well as integrating the *Domain Knowledge Model* of the legacy system with the traces.

  (c) Finally, it represents the task semantics in a well-defined model, i.e. *Semantic Task Model*, which is suitable for supporting the automatic GUI generation.

In addition to modeling the existing task, this process can also extend task modeling in cases where the desired task involves using multiple systems. **URGenT** accomplish this by

(a) analyzing the extended functionality that uses multiple systems,

(b) integrating the extended functionality with the existing task in the legacy system, and

(c) developing an integral model of the extended task.

Furthermore, **URGenT** provides to users a browser for validation of tasks' inferences. with the assistance of an understandable viewer, which shows the expert user the analyzed result of the task semantics, **URGenT** is able to get the confirmation from the user. The analysis result can also be collected as part of the domain knowledge. The strategy for task analysis and the structure scheme for both task model and domain knowledge model will be discussed in detail in the chapter three, "Task and Domain Modeling".

- *GUI Development*

  To support the automated generation of the new GUI, the following steps are used in **URGenT**.

  (a) First, the *Semantic Task Model* is transformed into a *Conceptual Interface Model*,

  (b) Then, a scheme of interface layout and graphical objects selection rules is selected, which is stored as a part of the *Domain Knowledge Model*.

  (c) Finally, a GUI is developed automatically out of the *Conceptual Interface Model*, based on the selected design rules that is related to the different user profiles.

  The detail of this process is introduced in the chapter four, "GUI Development".

The rest of this chapter is organized as follows. First, it presents the system architecture of **URGenT** and describes it in detail. Then, it briefly introduces a generic architecture for interface migration. Finally, it concludes by comparing the **URGenT** architecture with the generic framework and discussing their relationships and differences.

This chapter uses the following convention for all the diagrams, in order to maintain consistency.

(1) round-circled boxes represent the processes executed in the system,

(2) ellipses represent the intermediate products of the system processing,

(3) rectangles represent the existing systems or tools,

(4) rectangles with gray background represent the final products of the system,

(5) arrows represent the directions of the information flow,

(6) polygons represent the processes that requires human interaction.

2.1 Overview of URGenT Architecture

URGenT is the prototype tool that supports *Interface Migration* of the legacy system, which develops the GUI that improves the usability of the legacy system and extends its existing functionality to some degree. It accomplishes *Interface Migration* by three implementation stages. The overall architecture for **URGenT** is depicted in Figure 2.1.

- *Trace Recording*

  Traces are recorded interaction by the middleware when users actually perform the existing task with the system interface. Based on the trace information and domain knowledge, the *Trace Recording* stage recovers the original interface requirements and design decisions for the specific task at hand. The *traces* can also be extended with the interaction performed with multiple applications, therefore support extension of tasks in the legacy systems.

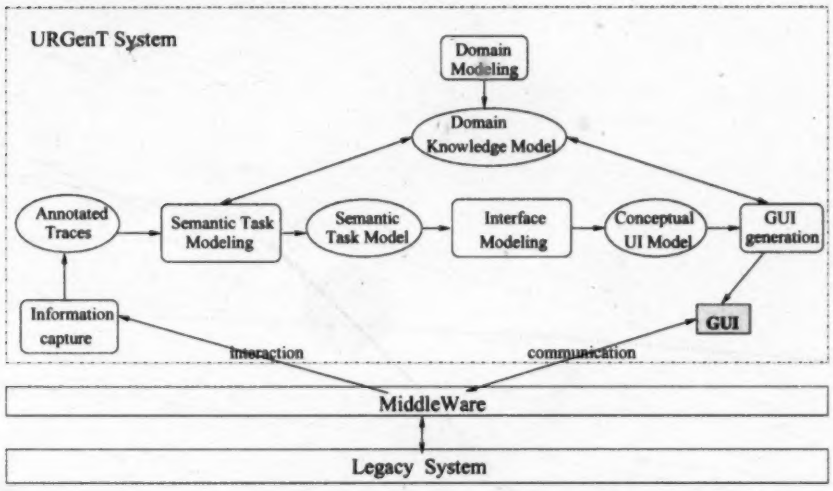

Figure 2.1: Framework of Interface Migration in **URGenT**.

- *Task and Domain Modeling*

  The *Task and Domain Modeling* stage uses well-defined forms to represent task semantics and domain knowledge. Task semantics is represented in terms of the classification of the information flow, while domain knowledge is represented in terms of the attributes of the information objects. The *Semantic Task Model* is the basic element to support future *GUI Development*. The *Domain Knowledge Model* facilitates future analysis and interface design.

- *GUI Development*

  The *GUI Development* stage generates the task-centered GUI based on the *Semantic*

Task Model and interface design rules. It is achieved by two steps: *Conceptual Interface Specification* and *Model-Based GUI Generation.*

(1) *Conceptual Interface Specification* transforms the *Semantic Task Model* into a *Conceptual Interface Model*, based on certain rules of mapping, and

(2) *Model-Based GUI Generation* generates the target GUI as the front-end of the legacy system, based on the *Conceptual Interface Model* and a set of guidelines of interface design for different classes of users.

On the new GUI, the interaction with the legacy system is optimized by hiding irrelevant information and avoiding redundant operations.

The expert user participates in the stages of both *Task and Domain Modeling* and *GUI Development* to support task analysis and interface design. The thesis will discuss human interaction during the **URGenT** process in the following chapters that describe **URGenT** in detail.

2.2 MiddleWare Introduction

As it is introduced in the first chapter, **URGenT** relies on the middleware to communicate with the legacy system. The middleware is the **CelWare** toolkit developed by CEL Corporation [6]. The **CelWare** toolkit is composed of Dynamic Link Libraries (DLL) that draw and run various controls. It contains

(a) **Recorder**, which is used for the recording of the system-user interaction, while the user is actually performing the task.

(b) **CelPilot**, which is used by the external interface to control the legacy system, i.e. exchange information between the *Legacy Interface* and the external interface as if the user is operating directly on the *Legacy Interface.*

Figure 2.2 depicts the functionality of these existing tools and their interaction with our system. This section will give a brief introduction of their functionality and terminology.

- **Recorder** is an enhanced terminal emulator that records the system-user interaction in terms of the domain model. Its output from the collection of the daily usage of the legacy system is a directed graph, which maps the legacy system's screens and the transitions among them, henceforth called the *Interface Graph.* In the graph, the nodes correspond to the individual screens of the system and the edges correspond to the user actions that enable the screen transition during the navigation.

  (a) *trace* is the output from **Recorder**. In **Recorder**, one walk-through of a task is recorded in one *trace.*

  (b) *action* is used to describe the interaction that causes one screen jump to the other screen. In a task performance, a set of *actions* happen in sequence to accomplish the

task.

(c) *action item* is either the KeyStroke at each field level (i.e. the user's data entry on the screen or the special function key pressed for screen navigation), or the MouseTrack (i.e. the highlighting performed by the user to show the interested area on the screen). Each *action* consists of one or more *action items*.

As a result, a "map" of the system interface is developed from recorded traces for different tasks of the legacy system. Based on this "map", **Recorder** is able to identify the screens' snapshots contained in the trace as instances of unique system screens. The meta-data contained in the "map" comprises all the *screens* in the legacy system and *actions* that represent all the possible screen navigation. The legacy interface is viewed as a collection of *screens*. Each screen navigation is caused by one *action*, comprising a sequence of primitive interactions between the user and the screen, i.e. *action item*. An example of one trace recorded by the **Recorder** for a "claim report" task is shown in Figure 3.1 of next chapter.

- **CelPilot** is the runtime component of CelWare. It can connect with the legacy system, exchange information between other applications and the legacy system, therefore support communication between them. **CelPilot** can translate the user action to equivalent set of elementary action items applied to the legacy system's original interface. Therefore it can be used by other application, such as a designed front-ending GUI, to control the system for desired task performance. In conclusion, **CelPilot** enables other applications to access any data on any screen of the legacy system without programming, as well as the seamless integration of the new interface with the legacy system without any change on the system. Therefore, the user can accomplish the same task with the new interface, without any change to the legacy system.

2.3 URGenT and MiddleWare Interactions

URGenT supports the migration of the interface from a text-based, system-oriented (i.e. generic) design, into a graphical-based, task-centered design, i.e. one interface designed for each particular task. It adopts a three-tier architecture in which itself is the upper level, and the **CelWare** is the middle level that is used to communicate between the upper level, **URGenT**, and the lower level, the legacy system. This three-tier architecture is depicted by Figure 2.2.

All the **URGenT** system has to do is listed as follows.

- First, it uses as input a small set of recorded user-system interaction traces of a particular task that is recorded by the **Recorder**, and recognizes a screen navigation and information exchange plan that the user carries out to accomplish her task.

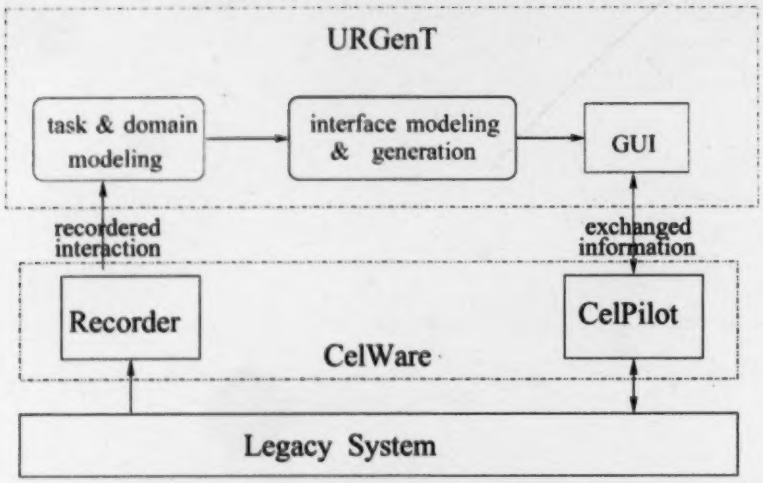

Figure 2.2: Relations Between the legacy system, The CelWare Toolkit, And **URGenT**.

- Then, it generates an abstract specification of the user interface, which should convey properly all types of information that need to be exchanged between the user and the system.
- Finally, given this specification, interface design guidelines [4], and a profile of the users that perform the task, it can generate a front-ending GUI, which uses the **CelPilot** to manipulate the legacy system in order to perform the desired task.

As you may notice, the first and the last steps, in which **URGenT** exchanges information with the legacy system, are accomplished with the help of the middleware, **Recorder** and **CelPilot**, respectively.

2.4 A Generic Interface Re-Engineering Architecture

This section will introduce a generic framework for interface re-engineering [27], and conclude the feature of **URGenT** by comparing it with the framework. This framework is depicted in Figure 2.3. Several problems as follows are addressed with the framework.

- How to understand the information and the process logic the current system interface embodies?
- How to represent both the static view and the dynamic behavior of the *Legacy Interface*, and make them work together for interface generation?
- How to generate the target interface that manipulates the legacy system?

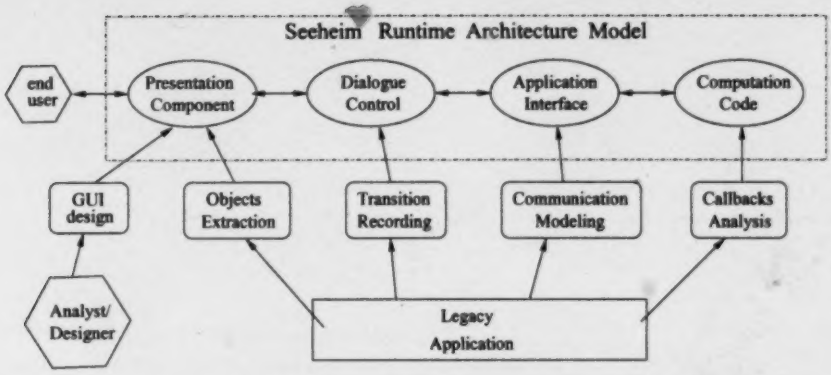

Figure 2.3: Generic Framework for Interface Modeling.

As you may find out from the figure, this framework focuses on the understanding and representation of the interface, in order to correctly interpret the existing interface. It understands the system from code, and is aimed to support the development of a functionally equivalent interface. This framework supports representing the user interface in a *Seeheim Model* [27], which defines a runtime architecture for a system that separates the computational code from the details of the user interface. The *Seeheim Model* allows the designer to specify the interface from different views separately out of the code.

This framework requires the extraction of four different views of the interface model from the original legacy code.

- The *Presentation Component* contains the content and visible state information of the system screens. It represents the static view of the user interface.
- The *Dialog Control* contains the screen states and transitions among screens of the system. It provides the dynamic information for the system interface to be traversed.
- The *Application Interface* is the communication channel between the application and the user interface. It computes the value and display message of the interface, and maintains the relationships between application data and interface objects.
- The *Computational Code* is everything left in the system after extracting interface model. It is maintained because its structure may be reorganized to fit for the callback form of the new interface.

The *Seeheim Model* is the basis for *user interface management systems* (UIMS), and its representation can be transformed into a UIMS representation and thus automate the process of the interface generation. The four different views of the model specifies both the static presentation and the dynamic behavior of the interface. The *Seeheim Model* also defines a runtime architecture for the interactive system that separates the computational

code from the implementation of the interface, so that the interface can communicate with the system. Since the different views of the interface are separated from the legacy application, the user interface can be tailored, customized, and updated as needed without having to modify the computation of the system.

URGenT has several features similar with this generic framework, for example, they both separate the interface design from the system implementation. Since **URGenT** is using three-tier architecture, the components in **URGenT** are totally separated from the legacy system by the middleware. Therefore, **URGenT** doesn't need to consider how to integrate the generated interface code with the original system code, and how to make the integrated code work properly with the legacy system. The middleware makes the bridge between them and enables them to work together naturally.

In the thesis, different features of **URGenT** are concluded as follows, by comparing **URGenT** with the generic framework.

- *Task Centered Interface Migration.*
  While the generic framework is intended to support interface re-generation for the whole system, **URGenT** is focused on the optimization of the human-machine interaction with respect to one particular task. In other words, **URGenT** supports task-centered interface design, rather than the interface design generic to all tasks of the system.

- *Interaction-Based Interface Understanding.*
  In the generic framework, the method of understanding the interface is to extract interface code fragments from the legacy code. While **URGenT** understands the interface by studying traces, which is recorded when the user performs the task with the legacy system.

- *Information-Flow Specification of Behavior.*
  (a) The *presentation component* in the generic framework corresponds to the static view of the interface, i.e. the information and the state of the interface, while **URGenT** uses the *domain knowledge model* to describe relevant static view of the legacy system.
  (b) The *dialog control* in the framework corresponds to the description of the dynamic behavior of the interface, i.e. the transition between screens and sequential relationship between interface objects, while **URGenT** extracts the information flow from *interaction trace* to represent the dynamic behavior.
  (c) The *application interface* in the framework corresponds to the communication model between the system and the interface, while **URGenT** uses a middleware for communication without any modification of the underlying interaction with the system.

- *"Round-Trip" Modeling*

  The modeling process in **URGenT** is a "round-trip", which goes from concrete description (trace) to abstract representation (semantic task model), and then from abstract representation (conceptual UI model) back to concrete design (GUI). Therefore, interaction optimization can be achieved during the circulatory modeling, but special cautiousness must be taken to avoid information loss.

- *Model-Driven GUI Generation.*

  In the generic framework for interface development, the different views of application model can be used to produce a static layout of the interface, and the run-time control for the interface. In **URGenT**, a *conceptual interface model* is developed by mapping from the *Semantic Task Model*, which describes the task requirements for the interface, with the task-oriented domain knowledge. And this *conceptual user interface model* is finally used for the actual GUI generation.

- *User Participation.*

  User Participation is not explicitly represented in the generic architecture. In **URGenT** however, the user can participate in almost all the stages of the design in this architecture. The user shows the interaction by actually performing the task, confirms the result of the task analysis, and participates the look-and-feel design for the real GUI. The approach of user participation in the task analysis and interface design will be introduced in both the chapter three, "Task and Domain Modeling" and the chapter four, "GUI Development".

2.5 Evaluation Criteria for Thesis Work

This thesis aims to support the graphical interface design that should be tailored to the tasks of the system's users. We call these interfaces as Task-Centered GUIs. Up to now, the main components in the **URGenT** system and the relationships between these components are briefly introduced. In order to accomplish the target to migrate task-centered interface, several standard requirements are brought up to be the criteria for developing the system.

This section will describe the criteria from two different aspects. Four requirements are applied for the design of graphical and task centered user interface, while three requirements are used against the prototype system we developed to support the *Interface Migration*.

2.5.1 Criteria for GUIs of URGenT

Four requirements are introduced here for the **URGenT**ś GUIs. The first one is the essential and most important one, and the rest three are the requirements for good GUIs.

- Correctness:

  The target interface, which gets rid of overlay operations, and only asks users for the dynamic data that has value local to each task performance, should make sure of its functionality equivalence with the original interface, and enable users to perform tasks appropriately with it.

- Efficiency:

  Task-centered interface design makes it possible to organize the relevant information in a set of screens smarter. In other words, the target interface should hide any information not relevant to the task, and at the same time, optimize the interaction of the related tasks as much as possible by the following certain rules, such as avoiding the iterative operations, entering fixed-value data to the system without asking input, and so on.

- Usability:

  A graphical user interface, which takes advantage of easy-to-understand graphical objects, should be intuitive to the user's view and easy to use, therefore it may save the training time of users. Moreover, by deploying some good functions the graphical interface provides, such as copy and paste, the target interface should facilitate user operations.

- Consistency:

  A consistent design throughout one system makes it easier and more predictable for the user to follow, therefore reduces the efforts of the user in trying to follow the style and arrangement of the interface and allows the user to only concentrate on major tasks presented by the system.

2.5.2 Criteria for URGenT

For the prototype system, **URGenT**, two requirements are brought up as its criteria.

- Extensibility:

  With new technologies keep growing, there is little advantage if the target interface only accomplish the existing functionality of the *Legacy Interface*. Therefore, the method developed in the thesis is aimed to generate the interface that is capable of working with other applications to extend functionality of the existing task.

- Generality:

  The prototype support should be general and helpful to the design of task-center graphical user interface for any legacy system. In other words, no assumption should be made on the underlying system. Also, the prototype system of the thesis should

provide a set of guidelines on the interaction objects and processes that the target interface should support, not the look-and-feel of the interface, in order to give the designer the space for creativity.

Chapter 3

Task and Domain Modeling

As introduced in the last chapter, the **URGenT** system involves two consecutive processes: *Semantic Task Modeling* and *GUI Development*. This chapter will give a detailed description of the first process in **URGenT**: *Semantic Task Modeling*, which consists of *Task Modeling* and *Domain Modeling*. Since the thesis uses the same terms to describe models, in order to keep the consistency between those models, you may see the same terms appear at several places for different model description in this chapter.

- Task Modeling

  Task Modeling aims at understanding the task functionality and results in the task representation at an abstract level. The first step of the *task modeling* is the *Understanding* step, which analyzes the information and the process logic that the *Legacy Interface* embodies. It can support the development of the new interface to replace the existing interface as the front end of the legacy system, and ensure its usability and its functional correctness. Once the *Understanding* step identifies all the information components and their relationship within the task, a *Modeling* step follows to develop a *Semantic Task Model*, which is an abstract specification that describes the data exchange for the task functionality. The thesis names the two major steps as *Traces Understanding* and *Semantics Modeling*.

  1. Traces Understanding

     This step involves detecting how the system is currently being used in the existing tasks, based on the task-specific traces that are recorded by the middleware. The following questions are addressed in the *Traces Understanding* stage.

     (1) How to capture enough information of the current interface in order to understand the functionality of the task?

     (2) How to acquire a navigation plan that accomplishes the task in the text-based interface of the legacy system?

2. Semantics Modeling

   This step involves the approach of representing the detected task semantics. There are two questions that must be answered for *Semantic Task Modeling*.

   (1) What content should be included in the task model, to represent sufficiently the requirements for the task?

   (2) What is the appropriate scheme for the task model, which may facilitate the generation of the interface model in the future?

- Domain Modeling

  Domain Modeling is the process of developing and increasing knowledge about the system and its tasks, and documenting the knowledge in a formal way in order to facilitate its future usage by *Task Modeling*. Since the inference of task semantics can be vague or even incorrect without the knowledge of the application domain, it's necessary to collect and increase the domain knowledge during **URGenT** implementation. In the future, the domain model can be used to annotate the task-specific traces, therefore optimizes the development of *Semantic Task Model*. The domain knowledge is composed of the objects in the system, their attributes and the relationships between them. The detail will be introduced later with an example.

3.1 Traces Understanding

This step takes as input the traces recorded by the **Recorder**. Each trace records one performance of the task at hand, i.e. what actually occurs while the user accomplishes her task with the *Legacy Interface*. The target is to extract the following information of user tasks from those task-specific traces.

- How the users navigate the text-based legacy interface to solve their problems (i.e. tasks)?
- What information the users exchange with the system during the task performance?

Since most legacy interfaces deal with simple alphanumeric display screens, a quite complete and explicit understanding of tasks in the domain may be developed by simply observing user input and system display during task performance. User input can be recorded naturally, however, since the information of interest has to be told by the user, **URGenT** asks for a special action item, i.e. "highlighting", to be performed by the expert user, in addition to the actual interaction with the legacy interface for the task. That is, during the task performance for trace recording, all the information of interest, such as the output information on the final report, or the derived data obtained in the middle of the interaction, has to be told by the expert user who uses a mouse to highlight the field that holds the information. Although

it seems a little intrusive to ask additional action items, it is not that bad for the following reasons.

(1) First, it is required only for the expert user when she **defines** the task by performing it, not for the real user during his task performance.

(2) Second, it is just a simple mouse movement over the display field, which should be paid attention by the user in order to achieve the task. Therefore this requirements shouldn't be too much for the expert user either.

From the perspective of entities and events defined by Eva [13], the components of each trace recorded for the task in the system are classified as follows. All the components together provide the dynamic view of the system interface navigation for the task, therefore constitute the necessary sources for analyzing the task requirements.

- Entity:
  Each *screen* the user navigates to can be defined as an entity, i.e. "Something about which the system needs to hold information".

- Event:
  Each *action*, which is performed on one screen and causes it navigate to the other screen, is defined as an event, i.e. "Something which happens in the world that causes a change in the value or status of a data item or entity in the system".

- Element:
  Each *action item* is one actual system-user interaction, either the keystroke by the user, or the information provided by the system response. Since one or more *action items* build up one *action*, they are treated as the elements of the navigation.

3.1.1 Recorded Traces

At the beginning of the thesis, a hypothetical task is introduced, which develops a report in an insurance information system. In the following discussion, the insurance system is called "report" system, and the task is called "claim report". Consider a situation where the insurance companies computerized their claims department separately from their customer's database. Therefore there are two separate subsystems, namely subsystem1 and subsystem2, which contain information on the customers and their accident claims respectively. To generate the final report for the "claim report" task, the data of the accident claim can only be retrieved from subsystem2, while the claim numbers according to each client name are only searchable in the subsystem1. Suppose the lawyer doesn't remember the claim number, but only knows the client name. Therefore, in order to finish the task, she has to go into the subsystem1 first, and find out the claim number. Then she goes to subsystem2, enters the claim number she just got, finally retrieves the information needed to develop the

report. Since the text-based interface of the legacy system does not cache information and transmit it later for the user, she may have to do some iterative operations, such as going back and forth between screens, entering the same information several times during one task performance. Figure 3.1 briefly shows how the lawyer accomplishes the report task in the legacy interface. In this figure, the following legend is adopted:

(1) the round-corner rectangle represents each system screen;

(2) each screen name is marked on the top of each screen;

(3) the gray rectangle within each screen represents each field for user input;

(4) the name for each input field is marked to the left of the field;

(5) the arrow between screens represents the data flow and the direction of screen navigation.

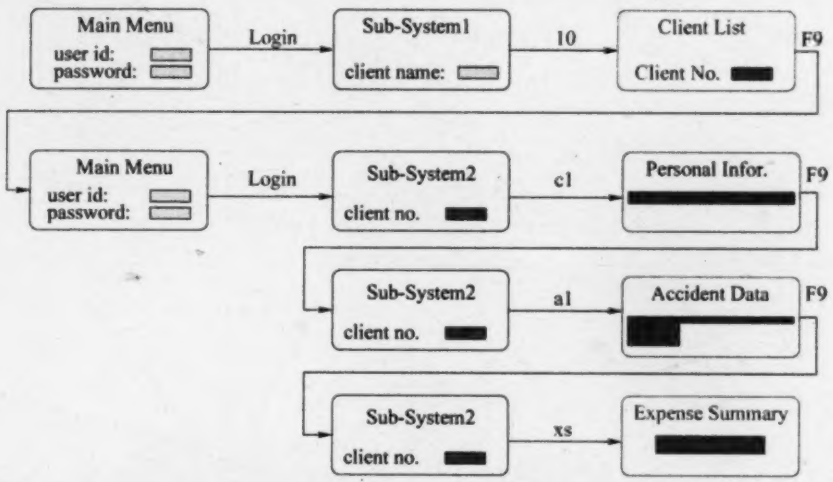

Figure 3.1: Task "Claim Report" in The Legacy Interface.

In Figure 3.1, you may notice that after getting the claim number, the user has to go back to the main menu, login to subsystem2 and enter the claim number to search the relevant information. Also, in order to retrieve three different information, i.e. client information, accident data, and expense summary in the "claim report" task, the user has to enter claim number three times in the menu screen of subsystem2. Table 3.1 depicts one trace of the "claim report" task recorded by the **Recorder** in this scenario. In the table, "screen name" depicts the screen navigation, "action" and "sequence id" tell the order among the actions, while "keystroke" and "mousetrack" are used to represent the data retrieved from user input or highlighted area on the screen.

URGenT takes as input several recorded traces of the same task, each of them is similar to this trace in Table 3.1, except for the problem-specific data that is particular to each task performance. Notice that because of the space limit, Table 3.1 only lists parts of

Screen Name	Action	Sequence	KeyStroke	MouseTrack
(1)*splash*	1	1	"*imsg*"@*E*	*None*
(2)*logon*	2	2	"*lanyan*"@*T*	*None*
logon	2	3	"*t65j*"@*E*	*None*
(3)*transaction*	3	4	"*subsystem*1"@*E*	*None*
(4)*name signon*	4	5	"*t65j*"@*E*	*None*
(5)*main menu*	5	6	"10"@*E*	*None*
(6)*inquiry search*	6	7	"*scott*"@*E*	*None*
(7)*claimant list*	7	8	*None*	(3, 12)*to*(4, 23)
claimant list	7	9	@9	*None*
(8)*main menu*	8	10	"12"@*E*	*None*
(9)*splash*	9	11	"*imsg*"@*E*	*None*
logon	10	12	"*lanyan*"@*T*	*None*
logon	10	13	"*t65j*"@*E*	*None*
(10)*transaction*	11	14	"*subsystem*2"@*E*	*None*
(11)*query menu*	12	15	"7889"@*T*	*None*
query menu	12	16	"*a*1"@*E*	*None*
(12)*accident*	13	17	*None*	(5, 4)*to*(6, 19)
accident	13	18	@9	*None*
(13)*query menu*	14	19	"7889"@*T*	*None*
query menu	14	20	"*c*1"@*E*	*None*
(14)*claimant info*	15	21	*None*	(3, 6)*to*(14, 22)
claimant info	15	22	@9	*None*
(15)*query menu*	16	23	"7889"@*T*	*None*
query menu	16	24	"*xs*"@*E*	*None*
(16)*expense summary*	17	25	*None*	(8, 9)*to*(9, 22)
expense summary	17	26	@9	*None*
(17)*query menu*	18	27	*None*	*None*

Table 3.1: One Task-Specific Trace Recorded in **Recorder**

attributes of each piece of data, i.e. the screen, action, sequence and content information of each data, but leaves out the location information recorded in the trace. In order to make the information simple and understandable to the reader, Table 3.1 uses the *Screen Name* instead of the unique *Screen ID* that is recorded in the **Recorder**. Besides, a special scheme for representing the data is adopted in Table 3.1. The *KeyStroke* represents the data entry from the user. The *KeyStroke* value that begins with "@" is used to represent as the special constant, which is a shared constant across different systems. For example, "@9" means function key PF9, "@T" means TAB key, and "@E" means ENTER key. Meanwhile, the *MouseTrack* represents the highlighted area that contains the information of interest on the system screen. "(x1, y1)to(x2, y2)" means that the information appears on the current screen with starting and ending coordinates (x1, y1) and (x2, y2) respectively.

Therefore, according to each number that is marked beside the screen name, the interaction procedure recorded in the trace is explained as follows:

(1) on the "splash" screen of the "report" system, the user enters "imsg" to navigate to "logon" screen,

(2) on the "logon" screen, the user types in "lanyan" as user id, goes to next field on the same screen by typing TAB key, and types in "t65j" as password, then types ENTER key to navigate to "transaction" screen,

(3) on "transaction" screen the user enters "subsystem1" to navigate to "name signon", the login screen of the subsystem1,

(4) on "name signon" screen, password 't65j" is entered again to navigate to "main menu" screen of the subsystem1,

(5) on "main menu" screen, "10" is entered as the function selection to navigate to "inquiry search" screen,

(6) on "inquiry search" screen, "scott" as the client name is entered to navigate to "claimant list" screen,

(7) on "claimant list" screen, the user highlights the wanted claim number "7889", which appears in the area from (3, 12) to (4, 23), and presses function key PF9 to jump back to "main menu" screen,

(8) on "main menu" screen, "12" is entered as a function selection to exit the subsystem1 and navigate to "splash" screen again,

(9) now, the user repeats the action item from (1) to (2) in the *Legacy Interface*, and navigates to "transaction" screen the second time,

(10) on "transaction" screen the user enters "subsystem2" to navigate to "query menu", which is the menu screen of the subsystem2,

(11) on "query menu" screen, the user types in "7889" as the claim number, then goes to next field on the screen by pressing TAB key, then enters "a1" to navigate to "accident"

screen,

(12) on "accident" screen, the user highlights the area from (5, 4) to (6, 19) which contains the accident information of this claim, then presses function key PF9 to jump back to "query menu" screen,

(13) on "query menu" screen, "7889" and TAB key are typed again, then "c1" is entered to navigate to "claimant info" screen,

(14) on "claimant info" screen, the user highlights the area from (3, 6) to (14, 22) which contains the personal information of this client, then presses function key PF9 to jump back to "query menu" screen,

(15) on "query menu" screen, "7889" and TAB key are typed again, then "xs" is entered to navigate to "expense summary" screen,

(16) on "expense summary" screen, the user highlights the area from (8, 9) to (9, 22) which contains the expense information of this claim, then presses function key PF9 to jump back to "query menu" screen,

(17) finally, all the information the user is seeking has been obtained, and the task is accomplished.

3.1.2 Initial Task Model

The traces recorded by the **Recorder** capture all the action items, and the exchanged information. In order to automate the understanding, **URGenT** characterizes every data item appearing in the task-specific traces in terms of *Distinction, Content* and *Temporal Relations*. Their relationships are shown in Figure 3.2.

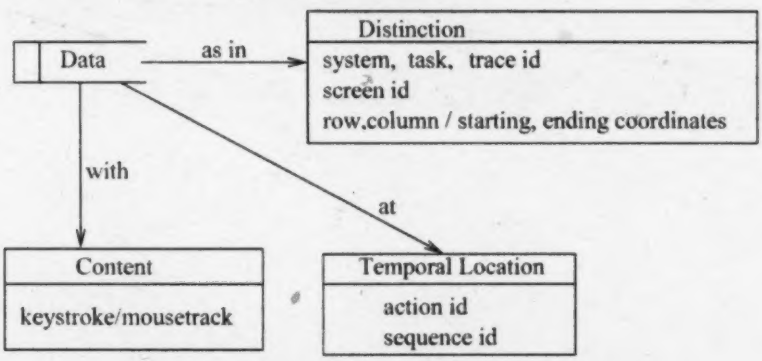

Figure 3.2: Relationship of Elements in Initial Task Model.

- **Distinction**: *System, Task Name, Trace ID, Screen ID* and *Row, Column* or *Starting Coordinates, Ending Coordinates*

  Since **URGenT** adopts the trace-based analysis method of task-centered *Interface*

Migration, each piece of data recorded should be in one distinct trace for a certain task of a system. And every recorded data appears at one particular location of the system. Therefore, each data identifies itself with the following attributes.

(1) *System* identifies the system with which the task is accomplished. In the example, "report" is the insurance system.

(2) *Task Name* is a unique name assigned to the task. In the example, "claim report" is the task we are discussing.

(3) *Trace ID* is the unique id used to identify the trace in which the data appears. In our example, suppose three traces of the "claim report" task are recorded for future analysis. Their trace Is could be assigned to three consequent and unique number.

(4) *Screen ID* is the unique ID of the screen in which the data is located. For example, the client name "scott" is entered in screen "inquiry search", which has unique screen id "11" in the system.

(5) For data input, by combining *Screen ID* with *Row* and *Column*, **URGenT** obtains its actual location. In the example, user ID "lanyan" is entered at row 30, column 8 on the "logon" screen.

(6) For highlighted data, by combining *Screen ID* with *Starting Coordinates* and *Ending Coordinates*, **URGenT** can locate its place precisely. In the example, the accident data can be retrieved from coordination (5, 4) to (6, 19) on screen "accident".

- **Content**: *KeyStroke* or *MouseTrack*

  Each data is exchanged by either data entry or highlighting movement over the data. Therefore its content is saved either in *KeyStroke* or *MouseTrack*.

  (1) *KeyStroke* stores the content of data entry, such as "xs" in the example.

  (2) *MouseTrack* stores the content of highlighting area. Notice that in Table 3.1, in order to save the space, the content in *MouseTrack* is replaced with the location information.

- **Temporal Relations**: *Action ID* and *Sequence ID*

  (1) *Action ID* is used to describe the order of screen transition during the trace navigation. In the example, the action 3 describes the event that causes the "transaction" screen jump to "name signon" screen, the third screen navigation in the trace.

  (2) *Sequence ID* is the occurrence time of this piece of data in the task trace, which represents the sequence dependency among action items. In the example, the action item with sequence ID 8 is the input of client name "scott", and it is the 8th elementary interaction of the task.

3.2 Semantics Modeling

This section is to answer the question of what is the meaning and the functionality of actions that are carried out to accomplish tasks? Generally, the tasks in the information system ask the user to provide to and obtain from the system some piece(s) of information. Therefore each task contains a combination of elementary information-exchange action items. **URGenT** focuses on the information-flow view of the data-oriented tasks in the information system. That is, **URGenT** adopts an "information exchange" analysis method, which is a high-level information classification, to understand the requirements for the tasks.

Three phases are required in using this method.

(1) The first phase is based on the analysis of one single trace of the task.

(2) The second phase is based on the analysis of multiple traces of the same task performed by the same user, who develops the same kind of reports for different clients.

(3) The third phase is based on the analysis of multiple traces of the same task performed by different users.

The functionality and result of each phase will be introduced after the definition of the data classification. After classifying the exchanged information in the task, **URGenT** uses a well-defined form to represent the task semantics. This task model should be able to facilitate the development of the target interface.

3.2.1 "Information Exchange" Analysis

URGenT uses "Information Exchange" theory to study the traces, that is, infer the task semantics by recognizing and classifying the exchanged information and how it is manipulated between users and the system. Based on the initial task model from the recorded traces, the ***"information exchange" analysis*** classifies the information flow of the task from two perspectives.

- Interaction Type: ***"tell"*** and ***"ask"***

  Inspired by the two basic discourse performatives in KQML [37], **URGenT** classifies actions into two types: ***"tell"*** or ***"ask"***, according to the direction of the information flow.

  1. *Tell* Interaction

     Action items for data-entry purpose are considered as ***"tell"*** interaction, i.e. the user provides information to the system. They are recorded as KeyStroke in the trace, such as "scott" – the client name – in the "claim report" task.

  2. *Ask* Interaction

     Alternatively, ***"ask"*** is data retrieval action item, by which the user obtains necessary information from the system screen. ***"ask"*** can be further divided into two

kinds of action items.

(a) *ask-standard* is always obtaining the information from fixed location on the screen, irrespective of the different data input in different traces. For example in "claim report", the claimant information is always retrieved from location (3, 6) to (14, 22) on screen "claimant info". and

(b) *ask-select* is searching the interesting information, which is located at different area on the screen according to different data input. For example in "claim report", the claimant number is retrieved from different place on the screen "claimant list", dependent of the client name entered by the user.

- Scope of Information: *System-specific* and *Problem-specific*

  There are special inputs with fixed values that are common across all the information systems, such as the TAB and ENTER keystrokes, which are called as "common constants" in **URGenT**. **URGenT** can recognize them even before any traces comparison. After that, **URGenT** classifies other different pieces of exchange information into one of two categories: *System-specific* and *Problem-specific* data, according to the scope of their value usage in the system.

  1. *System-specific* data has the same value or location across different traces of the task in the same system, independent of information exchanged in the task. In other words, it always has the same value (for data entry) or the same location (for display) across traces, no matter what data you have entered in this task performance. It includes *system constant, task constant, user constant* and *standard display*.

     (a) *system constant* is the data with the special value within one system, i.e. independent of which screen it appears on, and shared by multiple tasks in the same system. That is, it has constant value that is independent of the user's task when visiting these screens. For example, "subsystem1" is the fixed value that represents one subsystem in the "report" system, and in this system, it should always be entered at the special area in "transaction" screen, regardless of what task it is used for.

     (b) *task constant* is the special data with fixed value that must be entered in a certain screen in service of a specific task. Its value and location are fixed for this particular task. For example in "claim report" task, "10" on screen "main menu" is the selection for function of searching claim number by the client name in the subsystem1, therefore it should always be entered on this screen in order to accomplish the task.

     (c) *user constant* is the data that has value associated with the individual user

who performs the task, e.g. user id or password. Its value is unique and constant to each user, such as the user id "lanyan" for this particular user of the "claim report" task.

(d) *standard display* is the display information that always appears at the same location on the screen for a certain task, regardless of other information exchanged in the task. For example in the "claim report" task, the expense summary information is always retrieved from certain area on the "expense summary" screen, i.e. display from (8, 9) to (9, 22).

2. *Problem-specific* data has the value or location that is local to the individual trace, which may be different across different traces for the same task. It consists of *derived data, recurrent data, normal data* and *selective display.*

   (1) *Derived data* is defined as the input data retrieved from the previous system display, based on the assumption that the data input is either from knowledge of the user or from previous screen of the system. It comes from one system screen and is used as input in subsequent screens, such the claim number "7889" in the "claim report" example.

   (2) *Recurrent variable* is the data that the user entered several times on the same or different screens during the task performance. It may or may not have the same value across different traces of the same task, but it must have the same semantics in those traces. For example in "claim report", the claim number "7889" is entered three times in order to retrieve three different information of this claim.

   (3) *Normal variable* is the input data that only is entered once in each task performance, independent of any other input of the task. Its value does not have a direct source from previous system screen, therefore is assumed to be the knowledge coming directly from the user, such as the client name "scott" in the "claim report" task.

   (4) *Selective display* is the display information that appears at different locations of the screen across different traces, even for the same task. Its location is affected by other data input from the user. For example, in "claim report" task, the location for the highlighted area that contains the claim number is different, with respect to the client name entered previously and which claim record the user is interested in.

Using the "information exchange" theory, information classification on the recorded trace is accomplished in three successive phases. The result of each analysis phase of the "claim report" is depicted in Table 3.2. The first phase examines a single trace of the task, in order to create a baseline for the navigation graph that describes the user's traversal in the *Legacy*

Interface to accomplish the task in question, which is a sub-graph of the overall interface graph that will be developed by the **Recorder**. Each node in the navigation graph contains pieces of information exchanged during the task performance, and the edges of the graph demonstrate the navigation in the system interface. The first phase attempts to identify the different pieces of information that the user manipulates in the system. Based on the content of the different data elements entered and retrieved from the system, **URGenT** hypothesizes that data items with the same content are the same data object. As introduced before, some variables have special values that we can identify them as *common constant* from one single trace of the task, such as function key "@9". Recall that the **Recorder** classifies all the action items into *KeyStroke* or *MouseTrack*. Considering *KeyStroke* as "tell" action, and *MouseTrack* as "ask" action, the first phase of analysis can directly identifies *"tell"* and *"ask"* action.

Screen	Value	1st Phase	2nd Phase	3rd Phase
splash	*"imsg"*	$tell, var1$	*system – specific*	*system constant*
logon	*"lanyan"*	$tell, var2$	*system – specific*	*user constant*
logon	*"t65j"*	$tell, var3$	*problem – specific*	*user constant*
logon	*"female"*	$tell, var4$	*problem – specific*	*user constant*
transaction	*"name search"*	$tell, var5$	*system – specific*	*system constant*
main menu	"10"	$tell, var6$	*system – specific*	*task constant*
inquiry search	*"scott"*	$tell, var7$	*problem – specific*	*normal variable*
claimant list	$(3, 12)to(4, 23)$	$ask, var8$	*problem – specific*	*selective – display*
claimant list	@9	$tell, var9$	*common constant*	*common constant*
main menu	"12"	$tell, var10$	*system – specific*	*task constant*
transaction	*"report query"*	$tell, var11$	*system – specific*	*system constant*
query menu	"7889"	$tell, var12$	*problem – specific*	*derived & recurrent*
query menu	"a1"	$tell, var13$	*system – specific*	*task constant*
accident	$(5, 4)to(6, 19)$	$ask, var14$	*problem – specific*	*standard – display*
query menu	"c1"	$tell, var15$	*system – specific*	*task constant*
claimant info	$(3, 6)to(14, 22)$	$ask, var16$	*problem – specific*	*standard – display*
query menu	*"xs"*	$tell, var17$	*system – specific*	*task constant*
expense sum	$(8, 9)to(9, 22)$	$ask, var18$	*problem – specific*	*standard – display*

Table 3.2: The Classification Result from Analysis in **URGenT**

The second phase examines multiple traces of the same task performed by the same user, in order to differentiate the exchanged information between the *system-specific* and the *problem-specific* data. It identifies which data is *system-specific*, i.e. having the same value or location across different traces examined in this phase, and which are *problem-specific*, i.e. having the value local to the individual trace, which is different across traces. Traces of the same tasks may be non-deterministic, that is, the same task may be accomplished by alternative action sequences, thus the recorded traces may have different action and screen sequence. This assumption complicates the work of matching and comparing traces. To make work easier, **URGenT** assumes that all traces of the same task consist of the

same sequences of actions, therefore navigate to the same screens in the same sequence, i.e. they record deterministic task performances. Searching the method of analysis on non-deterministic task performances will be the subject of the future work.

Finally, in the third phase of the task analysis, **URGenT** proceeds to further classify the *system-specific* and the *problem-specific* data.

(1) First, **URGenT** classifies the *system-specific* data into four different types, i.e. *system constant, task constant, user constant* and *standard display. System constant* is the data that has the fixed value whenever it's used in the system, regardless of what task it is in, and what screen it navigates to. *Task constant* is the data that has the fixed value across different performance of the same task, as long as it appears on certain screens of this particular task. Comparing traces of the same task performed by different user, **URGenT** can distinguish *user constant* from other constant data that is independent of the user who performs the task, i.e. *system constant* and *task constant.* The data retrieved by ask-standard action is obviously the *standard display.*

(2) Then, **URGenT** further classifies the *problem-specific* data into one of the four possible types, i.e. *recurrent data, derived data, normal data* and *selective display. Recurrent data* has the same multiple appearances on certain screens in all the traces of this phase, but different values across traces. *Derived data* has value retrieved from previous system screen, not from the user's own knowledge. *Normal data* is the input data that is neither the *recurrent data* nor the *derived data* in the *problem-specific* type. And the data retrieved by ask-select action is obviously the data from the dynamic positions on the screen, i.e. the *selective display.*

3.2.2 Analysis Validation

After **URGenT** analyzes the information flow in the recorded traces, it classifies all the exchanged data into different types. However, the classification won't be a positive result before confirming it with the user. Therefore, **URGenT** also expects the user's confirmation to the analysis result, which is called *validation* in this thesis. There are two problems that should be addressed here.

- How to represent the analysis result clearly to the user?
- How to make it easy for the user to confirm the analysis result?

For showing the analysis result to users, **URGenT** produces a graphical representation of the task structure, in order to make the task navigation visualizable and reduce the probability of misinterpretations of the task. This graphical representation is called the *task viewer*, which describes the task in terms of its information flow. It shows visually the location in the *Legacy Interface* of each piece of identified data, i.e. on which legacy screen

and its coordination on the screen, as well as the circumstances that causes its appearance in the task, i.e. the navigation in system screens to accomplish the task. The nodes in the *task viewer* represent the legacy screens, and the arrows between nodes shows the direction path for navigation in the task. Each node has a list of objects in the screen with their identified data types. For the validation purpose, the *task viewer* has the upper view and the lower view as depicted in Figure 3.3, in order to show the navigation more clearly to users.

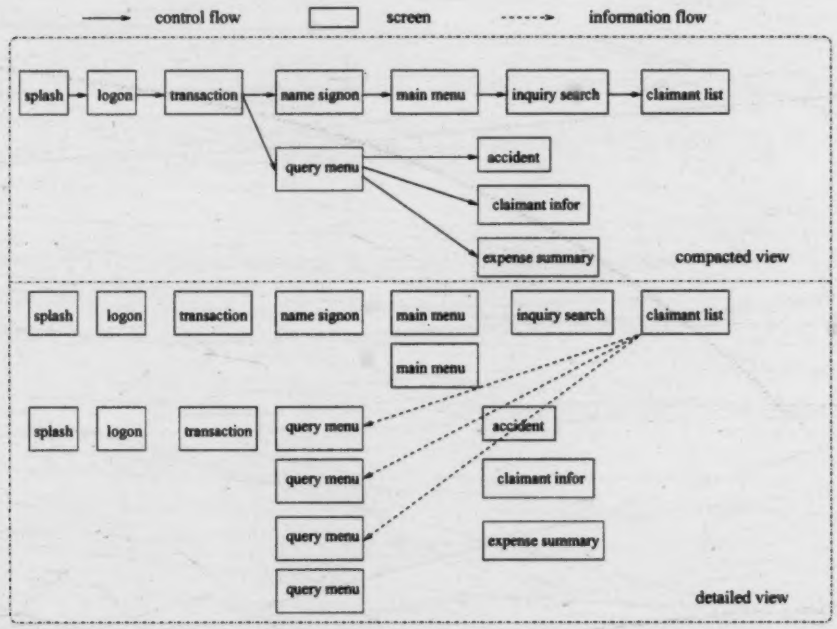

Figure 3.3: Task Viewer for Information Flow of "claim report".

- The upper view in the graph model demonstrates the screen navigation of the task. It represents every screen that appears in the task by one single node, even though it is visited several times. As a result, it clearly indicates how many legacy screens we should visit in order to get enough information for the task, and also gives the user an overall picture of the relationship between different screens. Each directed solid arrow between nodes represents an action that enables the screen navigation.

- The lower view shows the temporal relations between screens by using a left-to-right hierarchy with by-depth-first traversal method, which means that the screen sequence is represented from top to bottom. The directed dotted line between two nodes represents data flow, indicating the possibility that the earlier node contains the display that may be the source for the input data in the later screen. To every screen node

that is repeatedly visited during task performance, the lower view represents it by creating a new node in the same column but at the next row of its previous node. For instance, the screen "menu" has three nodes, it means that the "menu" screen is visited three times in sequence during this task performance.

In order to confirm the analysis result of a particular task, the *task editor* provides a list of data types with their proposed data in the task, and has a sequence scheme to conduct the user through the *validation* process. Every time when the expert user participates in the validation, she is lead through an entire question and answer procedure, and henceforth validates the analysis result from **URGenT**. When the user selects to validate one data object, the *task viewer* highlights the node in which the data appears to show its location in the task navigation, and pops up the legacy screen represented by this node to show the user the real appearance of the data on the screen. And it annotates each data object with its proposed data types. At this point, the user may approve the proposed data types as correct, in which case **URGenT** proceeds to the next step. Alternatively, the user may identify errors in the proposed data types, and make changes to the classification of data. Following is the sequence scheme for validation of different identified data.

- *system constant*

  When the data is possibly *system constant*, the expert user should be able to tell if this is a constant of this system just by its value. If it is confirmed, this data value is saved as the *system constant* in the *Domain Knowledge Model* in accordance with the system name. Otherwise, the data is left for the confirmation of task or user constant.

- *task constant*

  If the data is proposed as *task constant*, the viewer provides the expert user its value, screen and location where it appears in the task. If it is confirmed, this data's value is saved as the *task constant* in the *Domain Knowledge Model* with respect to the system, the task, the screen and its location. Otherwise, it may be *user constant*, and is left for the confirmation of *user constant*.

- *user constant*

  If the proposed data type is *user constant*, the viewer provides the expert user its value, as well as the screen and location it appears. If it is confirmed, this data's value is saved as the *user constant* in the *Domain Knowledge Model* with respect to the system and its location. Otherwise, it must be *problem-specific* data.

- *derived data*

  If the proposed data type is *derived data*, the viewer provides the expert user its value, screen and location where the data appears, as well as the location of the previous

display that contains the value. If the data type is confirmed, the user also points out which previous display is the source for this data. Otherwise, it must be *normal data*.

- *recurrent data*

  If the proposed data type is *recurrent data*, the viewer provides the expert user its value, all the screens and locations where its value appears. If the variable type is confirmed, the user also points out which screens and locations hold the same variable. Otherwise, it must be *normal data*.

- *derived & recurrent data*

  If the proposed variable type is *derived & recurrent data*, the expert user first judge if it is *derived data*, if it is denied, judge if it is *derived data*, if it is also denied, it must be a *normal data*. If both are confirmed, it is a *derived & recurrent data*.

- *standard display*

  If the highlighted data is analyzed as a *standard display*, the screen and location in which the data appears are showed to the expert user. If the data at this location on the screen is always interested, this area holds *standard display*.

- *normal data* and *selective data*

  All the input variables that are not confirmed up to now are assumed as *normal data*. Similarly, all the display data that are not confirmed up to now are assumed as *selective display*.

3.2.3 Task Modeling

Finally, **URGenT** adopts a method of *Semantic Task Modeling*, and represents the information flow of the task in a *Semantic Task Model*. In other words, after the data classification and *Domain Modeling* work together to understand task semantics, a well-defined representation of the task is developed, i.e. the *Semantic Task Model* based on the two orthogonal classification dimensions introduced earlier in the chapter. *Semantic Task Model* defines the user's task as a procedural pattern of information transactions, which consists of interaction objects with an associated set of attributes needed for the task. It describes the behavior of the interface independently of any interface technology, therefore it is especially useful to the interface migration [28].

The *Semantic Task Model* in this thesis describes what data the user should "tell" the interface and what information the user should "ask" from the interface. It expresses generally the functional requirements of the task and keeps it abstract enough to be independent of any interface technology. In the meantime, the *Semantic Task Model* is also able to map to any interface model, because its description is composed of interactive data objects which can be used as the interface objects. Optimization of system-user interaction, the

result of the abstraction that is achieved by the understanding of the exchanged data, will be explained in the chapter of **GUI Development**.

In this thesis, each task is represented by an association of particular sequence of input stimuli with corresponding sequences of output responses. The *Semantic Task Model* is composed of two sets of data in accordance with the data type, i.e. data in "tell" is the one entered by the user, and others in "ask" are the information coming from the system. Each piece of data in the model is specified in the same terms as in the captured initial task model, i.e. *Distinction*, *Content* and *Temporal Relations*. Although each term in the *Semantic Task Model* has different elements from the one in the initial model, the consistency makes the mapping process between models easy [18]. Relationships between the elements are shown in Figure 3.4.

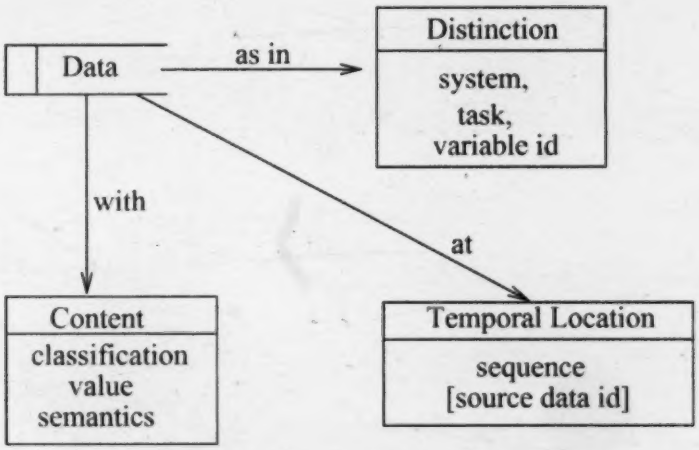

Figure 3.4: Relationship of Elements in Semantic Task Model.

- **Distinction**: (*system, task name, variable id*)
  Each piece of data is identified by the following attributes.
  (1) *system* is the name of the system in which the task is performed.
  (2) *task name* is the task in this system where this piece of data appears.
  (3) *variable id* is the unique id in this task to verify each piece of data.

- **Content**: (*classification, value, semantics*)
  These attributes give a deeper understanding of the interaction object, i.e. the meaning for each piece of data in the task performance.
  (1) *classification* classifies this piece of data into *system constant, user constant, task constant, derived variable, recurrent variable, normal variable, standard display* or *selective display*.
  (2) *value* is the content of this piece of data. And it is only stored as part of the

domain knowledge when it is system-specific data, because it is always the same in a certain task or in a system or for a particular user, which may be useful for the analysis in the future.

(3) *semantics* describes the meaning of this piece of data, which is part of the domain knowledge obtained from the expert user.

- **Temporal Relations**: (*sequence id, source data id*)

  These attributes define this piece of data's sequence relationship with other data in the *Legacy Interface*. They are used for the development of the target interface.

  (1) *sequence id* is the same as the *sequence ID* defined in the initial task model, and shows the object's sequential relationship in the information flow of the task.

  (2) *source data id* is only used by the *derived data*, and tells the unique data id of the previously displayed information that this piece of data is dependent of. In the target interface, this information has to be retrieved by the user before she is asked for the input of this data.

In Figure 3.5, several typical data represented in the *Semantic Task Model* are depicted to give a brief idea of the representation. Notice that,

(a) When the data is not a constant, i.e. the *value* of the data is different across multiple traces for the same task, the data is given a *value* as "None" in the model, because its value is not fixed across traces and should be provided by the user during the real interaction.

(b) When the object is a display, its *value* as "(x1, y1)to(x2,y2)" is the coordinations it appears on the screen.

(c) Any attributes with value as "–" indicates that it is unnecessary or unavailable right now.

As a result, from the *Semantic Task Model*, **URGenT** understands that in order to achieve the task, which information the system needs to get from the user and which information the system screen provides the user. According to the specification of the "claim report" task, data with *variable id* 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 15, 17, must be exchanged between the system and the user, in sequence defined by *sequence id* of each data. Now let's look at each specified data listed in Figure 3.5. **URGenT** understands from specification of data with variable id 2 that this data is a user constant and has a fixed value "lanyan". Therefore, as long as the user is the same person, the user doesn't need to provide this information to the system again. As for variable 7, it is a display message from the interface, and the location is from (3, 12) to (4, 23) on the current system screen. The specification of variable 12 tells that this data derives its value from display variable 7, and is entered three times in sequence 16, 20 and 24 of the task performance. On the examination of the *Semantic Task Model* for "claim report", it is concluded that the *Semantic Task Model*,

Representation for Task "claim report"

system :	report
task name:	claim report
"tell" variable id:	1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 15, 17
"ask" variable id:	7, 14, 16

Representation for Data in Task "claim report"

variable id:	2
system :	report
task name :	claim report
sequence id:	2, 13
semantics:	user id
classification:	user constant
value :	lanyan
source variable id :	–
variable id:	7
system :	report
task name :	claim report
sequence id:	8
semantics:	–
classification:	selective display
value :	(3, 12)to(4, 23)
source variable id :	–
variable id:	12
system :	report
task name :	claim report
sequence id:	16, 20, 24
semantics :	claim no.
classification :	derived data & recurrent data
value :	7889
source variable id :	7

Figure 3.5: Several Data Specification in *Semantic Task Model* for "claim report"

which describes the information flow of the legacy interface, is capable of supporting the target GUI development.

3.3 Domain Modeling

Domain Knowledge Model is the representation of a knowledge database that contains the important objects and their attributes in a given domain. It can be generated by identifying objects that are common to all applications within the domain, and storing the knowledge for attributes of all objects. It's often hard to decide the scope of the domain, i.e. what system should be outside or included in the domain. This thesis assumes that one domain contains just one single system.

As discussed in last section of the chapter, after the *information-flow exchange* analysis, **URGenT** knows how users exchange the information with the *Legacy Interface* to accomplish one task, but it still doesn't understand the semantics and functionality of those exchanged data. Aimed to solve this problem, **URGenT** collects the common knowledge about those exchanged data in the domain, uses a well-defined form to represent and store them, and annotates them during analysis process. These procedures are accomplished in the *domain modeling* process. This section will introduce the *domain modeling* that is relevant with the task analysis. Two aspects of domain knowledge are collected and deployed in the *Semantic Task Modeling*: the knowledge of *constant objects* and the *semantics for objects*. Two problems of the *Domain Knowledge Model* are addressed in both aspects.

- How to represent the knowledge in the *Domain Knowledge Model*?
- How does the domain knowledge help in the *task analysis*?

3.3.1 Constant Objects

In the discussion of analysis validation, it is addressed that every confirmed constant is saved as a part of the *Domain Knowledge Model*. However, since the value of each *problem-specific* data is varied across every performance even within the same task, it needs to be entered by the user every time a task is performed. Therefore, the knowledge of the *problem-specific* data is not part of the *Domain Knowledge Model*, which should only contain static knowledge of the system. The structure for the knowledge of confirmed constant and examples on the "claim report" task will be described in this section. After this knowledge is saved in the *Domain Knowledge Model*, in the developed GUI, when the system needs the same information with the fixed value, this value is entered without asking for user input.

The thesis classifies three types of constant data in every domain: *system constant*, *task constant* and *user constant*. The domain knowledge about those special objects can be increased by collecting their values with respect to the system and the task, because

(1) *system constant* has the fixed value within one system, which is independent of tasks and screens,

(2) *task constant* has the fixed value appearing on a certain screen within the same task of the same system, while

(3) *user constant* has the fixed value associated with a certain user.

In order to help the expert user to understand the task and data for validation, **URGenT** generates a graphical editor which is introduced in section 3.2.2, and asks the user to validate or deny the data classification by **URGenT**. After the data validation, *Domain Modeling* builds the *Domain Knowledge Model* incrementally by adding the knowledge of constants in the system. For the purpose of facilitating the analysis process in the future, the knowledge of *constants* are represented in the same terms as the *Semantic Task Model*, i.e. *Distinction* and *Content*. Domain knowledge is the static view of the system, and is not affected by the time, so no *Temporal Relation* is needed to describe it.

1. **Distinction**: (*system*, *task name*, *vid*)

   (1) *system* is the name of the system in which the *system constant* always has the fixed value.

   (2) *task name* is the name for the task within this system in which the *task constant* always has the fixed value.

   (3) *vid* is the unique id that is assigned to the constant object by **URGenT** within the particular task.

2. **Content**: (*value*, *classification*)

   (1) *value* is the fixed value for this constant object.

   (2) *classification* defines the constant as *system constant*, *task constant*, or *user constant*.

The knowledge of constant data in the "claim report" example is depicted in Table 3.3. Note that the column with the value "–" represents unnecessary or unavailable data.

system	task name	vid	value	classification
report	–	1	"*imsg*"	*system constant*
report	*claim report*	2	"*lanyan*"	*user constant*
report	*claim report*	3	"*t65j*"	*user constant*
report	–	5	"*subsystem1*"	*system constant*
report	*claim report*	6	"10"	*task constant*
report	*claim report*	10	"12"	*task constant*
report	–	11	"*subsystem2*"	*system constant*
report	*claim report*	13	"*a1*"	*task constant*
report	*claim report*	15	"*c1*"	*task constant*
report	*claim report*	17	"*xs*"	*task constant*

Table 3.3: Knowledge of Constants

As the knowledge about constant objects in the domain increased, time for the *task analysis* process can be saved by directly getting constant data type from the fixed values stored in the *Domain Knowledge Model*. For instance, "imsg", the *system constant* on screen "splash" can be distinguished by **URGenT** without any analysis.

3.3.2 Semantics for Objects

Since asking explicitly the semantics for each object requires additional work from the user, **URGenT** only asks the semantics of data objects that would appear in the target interface. This knowledge could be used to build the label text for each object in the target interface. After the data validation, *Domain Modeling* searches for the semantics of input data during *Semantic Task Modeling*, and builds the *Domain Knowledge Model* incrementally, which will save a lot of time and make **URGenT** much more efficient.

The input objects of *problem-specific* data have three types: *recurrent data*, *derived data* and *normal data*. Since their value is varied across different task performance, they need the input from the user every time the task is performed. Therefore, a friendly interface for the task should ask the user for their value, by attaching the semantics of the objects next to their input area. Therefore, the knowledge about the objects' semantics is collected for the target interface development. **URGenT** uses the same viewer as the one for the analysis validation, which presents the user the objects' appearance and usage in the information system for the particular task, and asks the expert user to provide semantics of objects, which are customized to represent the labels of those objects.

The *semantics for objects* are represented in the same terms as the *knowledge of constants*, i.e. *Distinction* and *Content*. And for the same reason, *Temporal Relations* is not needed here too. This kind of knowledge from the example of the "claim report" task is depicted in Table 3.4.

system	screen	location	value
report	*logon*	(3, 27)	*user id*
report	*inquiry search*	(26, 4)	*claim name*
report	*query menu*	(34, 8)	*claim no.*

Table 3.4: Knowledge of Variables Semantics

1. **Distinction**: (*system, screen, coordination*)

   Following attributes identify the object that has the provided semantics.

   (1) *System* is the name of the system in which the objects appear.

   (2) *screen* is the unique id of the system screen on which the object appears.

   (3) *location* is the location on the screen where the object appears.

2. **Content:** (*semantics*)

   Semantics is the semantics of the object, which will be used next to the input field to tell the user what the interface wants from the input.

Chapter 4

GUI Development

Since the legacy system is often built for mainframes, whose purpose is to support simple transaction-based task such as data entry and retrieval, it does not perform long and complex computations, but achieves the task by interacting often with its users. Most of the *Legacy Interface* has the field-by-field interaction, i.e. the user inputs a piece of data in one field, and goes to the next screen or the next field of the same screen, then continues data input [39]. This kind of transaction is repeated, so that the task can get enough information until it is accomplished. However, the user cannot go back to previous field when she wants to change the entered information. One solution to the problem is to build the screen-by-screen interaction that replaces the field-by-field interaction, by clustering several screen snapshots together into one unique system screen [8]. The screen-by-screen interaction enables the data on the same screen to be entered in any sequence, and stores all the data it gets without processing them. After all the input on the screen being completed, the user submits those stored data altogether, then proceeds to the next screen. Therefore, before the screen is submitted, the user is enabled to change any input on this screen as many times as she wants.

In the *forward engineering* process, i.e. *GUI Development*, **URGenT** restructures the interface to adopt the solution of migrating from the field-by-field to the screen-by-screen interaction. As a result, a task-centered GUI is developed as the efficient front-end of the legacy system. In the task performance, the new GUI interacts with the *Legacy Interface*, which acts as the bridge for exchanging data to and from the legacy system in certain sequence. The task-centered GUI offers the following advantages.

(1) First, it uses graphical objects, such as buttons, lists, combo boxes, to make it user-friendly, and provides the ability to cut and paste between screens [6], which eases the pain of typing every data in all fields.

(2) Second, it deploys the cache function to buffer data before passing it to the system, and releases the user from remembering every data and functional command for task navigation.

(2) Third, task-centered interface design is tailored to each specific task, therefore simplifies

the interaction of each task.

The automatic generation of GUIs from declarative descriptions can reduce costs and enforce design principles [36], and focuses the designer's attention on human activity and the benefits associated with the adoption of the new interface technology [33]. Having the target to automate the interface generation, *GUI development* of **URGenT** first generates a declarative interface specification from *Semantic Task Model*, then builds GUI from this specification. Therefore, it is composed of two major stages: *Conceptual Interface Specification* and *Graphical Interface Generation*.

4.1 Conceptual Interface Specification

Conceptual Interface Specification is an important intermediate stage in the transition from *Semantic Task Model* to interface implementation. It takes the *Semantic Task Model* as input, and generates an abstract specification for the target interface. A directed graph can be used to describe the *Conceptual Interface Model*. In the graph, each node characterizes the individual screen of the target interface by a bunch of interface objects which are independent of the platform, and each edge corresponds to the action that enable the transition from one screen to another. The following problems will be addressed here.

- What is the appropriate scheme to represent the *Conceptual Interface Model*?
- How can the *Conceptual Interface Model* ensure the functional correctness of the target interface?
- How should the *Conceptual Interface Model* plan and optimize the navigation of the text-based interface, so that an efficient interface can be developed based on the interface model?

The thesis is focused on the reporting tasks in the information system. The reporting tasks are basically exchanging information between users and the system, rather than planning the sequence of interactions. As a result, the ordering of actions in each task is not considered as critical [18], but it is crucial to decide what information should be exchanged to accomplish the task. The *Conceptual Interface Model* constitutes a bridge between the *Semantic Task Model* and the target interface. It should be abstract enough not to consider the real interface objects selection or layout strategy, yet have enough detail to support appropriate interface design and remain the functional correctness of the interface. Generally, the high-level interface model contains two aspects of the interface [31]:

- *presentation* defines the screens in the GUI and the interaction objects needed in each screen and

- *manipulation* defines the navigation among screens, i.e. actions performed on each objects, and the sequence of actions that are triggered by the user inputs.

Based on the assumption that the *Semantic Task Model* has captured the data flow and their sequential information in the task, **URGenT** generates the interface model by directly mapping from *Semantic Task Model*. In **URGenT**, the *Conceptual Interface Model* specifies all facets for the design of the interface, such as interface objects, and their behavior. In order to ensure the functional correctness of the target interface, the major concern of the mapping process is to carry forward the requirements captured in the *Semantic Task Model*, therefore build the *Conceptual Interface Model* with none or very little loss of information. Several typical objects of the *Conceptual Interface Model* for the "claim report" task are listed in Figure 4.1. Note that, value "–" indicates unnecessary or unavailable data.

The interaction in the target interface can be optimized by some strategies used in the interface model, such as hiding from users the fixed-value objects that are classified as the system-specific data in the *Semantic Task Model*. The mapping process from the *Semantic Task Model* to *Conceptual Interface Model* can be accomplished by two steps:

- *Interface Objects Classification*
  In order to support the mapping of all the objects from the *Semantic Task Model* to *Conceptual Interface Model*, the thesis classifies the interface objects into two categories: i.e. *external* objects and *internal* objects, corresponding to data objects in the *Semantic Task Model*.

- *Conceptual Interface Modeling*.
  Since the *Conceptual Interface Model* is developed from the *Semantic Task Model*, it is composed of a set of interaction objects mapped from the data objects in the *Semantic Task Model*, which describes their attributes, as well as the relations between them. *External* and *internal* objects in the *Conceptual Interface Model* are represented in different terms. For example, the *external* object is identified by its label, while the *internal* object is identified by its fixed value.

In the following sections, according to different interface objects, the two steps will be introduced respectively.

4.1.1 External Objects

External objects are the ones that MUST be mapped onto a set of graphical objects, such as buttons, lists, or sliders that can be manipulated directly by the user on the screen. In the target GUI [22], each graphical object represents the data whose value must be provided by the user. According to their data flow direction, *external* objects in the *Conceptual Interface Model* can be classified into *input* and *display*.

Conceptual Interface Specification

system :	report
task name:	claim report
external objects:	user id, passwd, name, one record, claim no., accident info., claimant data, expense summary
internal objects:	imsg, wbocomp, 10, @9, 12, /rcl, aces, a1, c1, xs

One Internal Object in Interface Specification

system :	report
task name:	claim report
sequence id:	1, 12
value:	imsg

Several External Objects in Interface Specification

system :	report
task name:	claim report
label:	user id
object id:	1
source object id:	–
type:	independent input
sequence id:	2, 13

system :	report
task name:	claim report
label:	–
object id:	4
source object id:	–
type:	dynamic display
sequence id:	8

system :	report
task name:	claim report
label:	claim no.
object id:	5
source object id:	4
type:	intermediate input
sequence id:	16, 20, 24

system :	report
task name:	claim report
label:	expense summary
object id:	8
source object id:	–
type:	static display
sequence id:	26

Figure 4.1: Conceptual Interface Specification for Task "claim report"

- **Input** is the information that MUST be "told" by the user. Since the *problem-specific* data have to be entered in every performance of the task, it is mapped into the *input* object. According to the data source, the *input* objects are further divided into two types: *independent input* and *intermediate input*.
  (1) *independent input* is the input that the user can type in without any information provided by the system, which can be mapped from *recurrent* or *normal* data in the *Semantic Task Model*.
  (2) *intermediate input* is the input that can only be entered by the user after obtaining some information from previous system screen during the interaction, which can be mapped from *derived* data in the *Semantic Task Model*.

- **Display** is the information that is provided by the system screen, which is "asked" by the user within the task performance. The thesis also divides the *display* into two types: *static display* and *dynamic display*.
  (1) *static display* is the information that always appears in the same area on one screen, regardless of other data exchanged during the performance, i.e. the *standard-display* from *Semantic Task Model*.
  (2) *dynamic display* is the information that comes from dynamic location on the screen, which is local to each performance of the task, i.e. the *selective-display* from *Semantic Task Model*.

Each *external* object is represented in terms of *Distinction*, *Temporal Relations* and *Execution*.

- **Distinction**: (*system, task name, label*)
  System, task name and *label* identify each object in the target interface. *system* and *task name* tell the system and the task where the interface object appears. *label* is the semantics of the interface object that may appear next to its input field in the target GUI so that the user knows what information is asked here.

- **Temporal Relations**: (*Object id, source object id*)
  Object id and *source object id* illustrate the status of each object and their relations in the target interface. *Object id* tells the order this object appears in the target interface, therefore indicates the temporal relations among objects. *Source object id* is used only when the object is *intermediate input*, in order to point out the *display* data source for this object.

- **Execution**: (*classification, sequence id*)
  The execution attributes basically describe how the target interface manipulates the system to accomplish tasks through these objects. The *classification* of the object may

be *independent input, intermediate input, static display* and *dynamic display*. Objects with different types are treated differently during the GUI implementation. *Sequence id* stores the information of how to deal with the object in the run-time interaction, i.e. when this input's value should be entered, or when the display can be retrieved from the legacy interface. This information is necessary during the GUI execution when the target interface caches and enters the data in the correct sequence, especially when the same data is asked multiple times by the system.

4.1.2 Internal Objects

Internal objects are used during the task performance with the legacy interface, but do not appear in the target user interface, such as the constants in the *system-specific* data of the *Semantic Task Model*. They are hidden from the target interface to optimize the task interaction, but remains in the interface model because the developed new GUI need to communicate with the *Legacy Interface* by passing all the data, therefore manipulate the legacy system to accomplish the task. For example the system constant, such as "imsg" in the Table 3.1, since its value is always the same, **URGenT** can enter its value automatically without asking the user from the target interface, therefore hide this object from the user in the target GUI. However, it has to be entered in the legacy interface to ensure the accomplishment of the task. The *internal* objects in the *Conceptual Interface Model* is to store this kind of information underlying the target interface and ensure that the legacy interface can get the data whenever it wants.

Each *internal* object is represented in terms of *Distinction* and *Execution*. Since *Temporal Relations* of objects are used for their layout location on the target screen, and *internal* objects will not appear on screen, it is not necessary to store this information for *internal* objects. The specification of *internal* objects prevents loss of information during the mapping process, and ensures the accomplishment of the task in the legacy system.

- **Distinction**: (*system, task name, value*)
  system, task name and *value* identify each object that is hidden underlying the target interface. The function of the *system* and *task name* is the same as that in the *external* object. *value* is the value for the object that will be entered to the *Legacy Interface* automatically by **URGenT**, without showing up in the target interface.

- **Execution**: (*Sequence id*)
  Sequence id is the ordering information to enter the object's fixed value during the run-time execution of the task.

4.2 Graphical Interface Generation

Consequently, *Graphical Interface Generation* is the process of developing from the *Conceptual Interface Model* the graphical interface that actually acts as the front end of the legacy system. It is aimed to create the executable GUI from the *Conceptual Interface Model*. Since a lot of coding effort is often devoted to the implementation of the user interface [28], it may save time and work load dramatically by automating the construction of the user interface to a certain degree. The following are three requirements listed in the order of importance for the target GUI.

- Executable
  It should be able to manipulate the system by exchanging data items between them with the assistance of the legacy interface, in order that the task can be actually accomplished.
- Correct
  It should ensure its functional correctness comparing with the original interface, by delivering "tell" data items from and feeding "asked" data items to the system.
- Optimized
  It should try to optimize the performance of the task, by asking users to input every distinct data item once, and buffering them for later delivery when they are needed.

Based on the discussion above, the following problems should be solved by this process.

- How do we automate the generation of the GUI from the *Conceptual Interface Model*?
- How does the newly developed interface interact with the system?
- What do we ensure the correctness of the new GUI and try to improve its performance?

To design an actual GUI that implements the user's task, the object appearance and the interface layout have to be decided at this point. Therefore, *Graphical Interface Generation* involves
(1) selecting the design style and graphical objects, in correspondence with the user profile,
(2) choosing the appropriate rules for interface layout, and finally
(3) developing the graphical interface tailored to a certain task and a class of users, based on those rules.
URGenT integrates the selection rules of interactive objects and a layout rule of GUI, and generates the guidelines for the GUI development. Having attached the guidelines, it proceeds to develop the GUI with appropriate graphical object for each information in the task flow. **URGenT** increases the quality and consistency of the resulting GUI by using an

inference rule to automate objects selection and layout of the GUI [22]. Four major issues are addressed in this section.

- *Graphical Objects Selection*
  Which graphical objects is appropriate to represent each interactive object that is specified in the interface model?
- *Interface Layout*
  How to localize and in which order should these objects be laid out on each screen in the target GUI, and with which dimension the objects should be displayed [14]?
- *Domain Modeling*
  What knowledge of the domain and their functionality can be collected during this process?
- *Run-time Execution*
  How to generate the actual GUI and enable the GUI to manipulate the legacy system for the task accomplishment?

4.2.1 Graphical Objects Selection

In *Conceptual Interface Model*, all the external objects (input and display) are the objects to appear in the target interface. The different data types of the objects imply the need for different graphical objects in the target GUI. There is a class of graphical interaction objects appropriate for each data type at hand [19]. For example, to the data that indicates a date, appropriate graphical objects might be a calendar, a combination of three scrolling lists for year, month and day selection, or a simple text entry box, etc. While to password entry, it's appropriate to select password entry field, which doesn't echo the input on the screen. Also, according the classification of objects function, such as input and display, the selections of graphical interaction objects to represent them are different. For example, the display is the information obtained from the *Legacy Interface* that should not be changeable, so the interface object for it can be a label or a text area. Figure 4.2 shows one screen of the generated GUI for "claim report" task.

URGenT also considers the standard of graphical objects selection differently according to the user profile, which contains different classes of users. Different users prefer different objects selection. For example, users with poor typing skill may prefer a list of possible values for selection to a text field for typing. Users can be classified into many different types in terms of their abilities, knowledge and preferred style of information processing. In the thesis work, **URGenT** has two preliminary user profiles that classify users into two types: expert users and novice users. Different assumptions about the preference to the graphical objects are made on the two types of users:

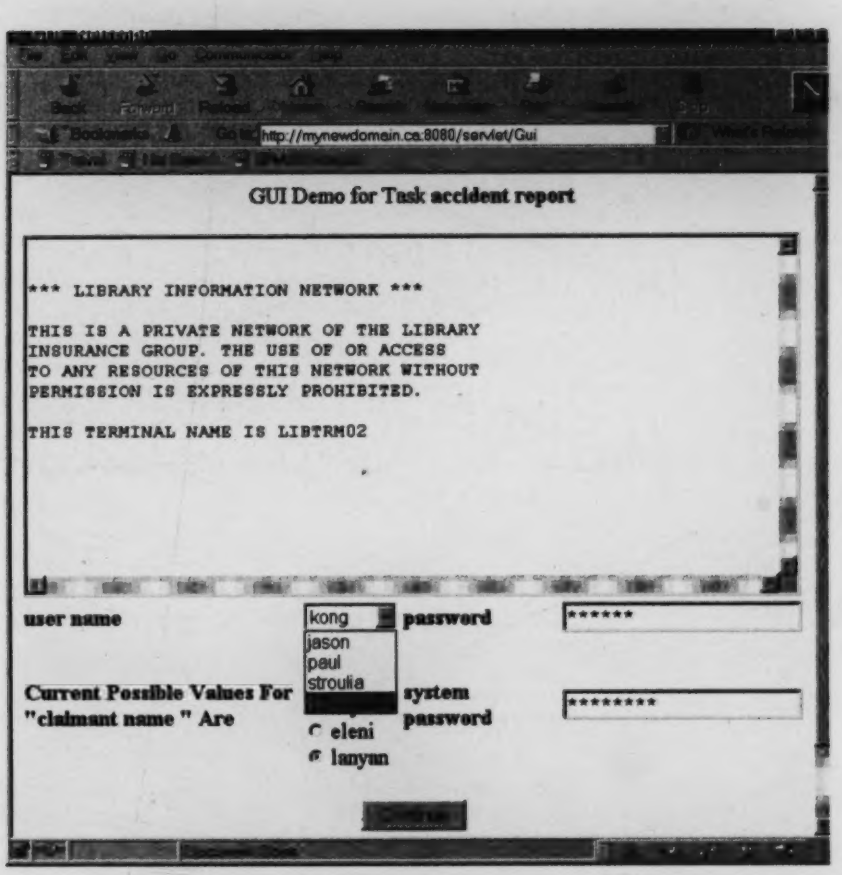

Figure 4.2: One Screen in The Developed GUI for "claim report"

- Expert users are familiar with system and task, and they have authority to enter new data into system during interaction, therefore, they prefer typing in data directly to selecting data from long list. For example, **URGenT** provides input text fields as default objects for input from expert users.

- Novice users have little knowledge of the system, the task and the interaction method, so they want the interface to provide as much information as possible. For example, **URGenT** selects combo boxes or scrolling lists, which list all the possible values of objects for selection, as default objects for input from novice users.

As a result, current **URGenT** can either automatically assign the graphical object to each data item, according to the user type as well as the data types and objects classifications, or let the user participate in the graphical object selection, by presenting several choices for the user, and annotating the user's selection to the interface model.

4.2.2 Interface Layout

The *Interface Layout* takes a set of interaction objects and a parent window as input and produce a spatial arrangement of the objects within the window as output [22]. Since we are dealing with small tasks, in terms of data objects and actions, the issues of getting the knowledge of how to place objects in screens are not very complex. The first issue in the interface layout is the ordering of objects to appear on the screen. Based on the assumption that the input sequence of the legacy interface ensures the correct task performance, and is natural to the user, the target GUI is designed to ask input in the same sequence as the legacy interface for manipulating the system properly.

Currently, **URGenT** places all the *independent input* into the start screen, and asks for each *intermediate input* only after its source display has been showed in the target interface. Therefore, from the specification of external objects in the *Conceptual Interface Model*, the knowledge of how many screens constitute the new GUI and what objects are represented on each screen can be deducted by reasonable inference. There are two kinds of strategies for objects placement [14].

- *Dynamic placement* allows a see-and-place computer-aided generation, but this strategy distracts designer from semantic design by asking her to pay too much attention in the look-and-feel of the interface.

- *Static placement* is to localize objects either in a two-column or a grid-based structure. Since it can be achieved automatically and relieve the unnecessary burden for the designer, the thesis adopts the two-column strategy in the *Static placement*.

In the real GUI development, the dimension of each object in the screen also should be decided. **URGenT** uses the fixed two-column placement, i.e. each row on the screen has

two input objects, therefore makes it easy to decide their dimensions according to the screen size. The *semantics* for all input data are treated as the static objects in the target interface, and should always be shown next to its input area. Therefore, **URGenT** arranges them right-aligned to the left of the graphical objects for their input objects [22].

4.2.3 Model-based GUI Design Issues

Domain knowledge of the interface design can be built incrementally during the implementation of **URGenT**, therefore not necessarily from scratch. The acquisition of new knowledge is aimed to keep up with changes or further improve the efficiency of the target GUI. Collecting attributes of the interface objects in a domain is helpful for the future GUI design. For example, integrating the data type of each piece of information with the selection rules can help automating the graphical objects selection, and supporting optimal usage of the GUI advantage.

Since different users have different skill and style for the task performance, more sophisticated application of the design rules may be explored to meet some users' need. The creation of interfaces is accomplished from a set of pre-defined task-specific interactive objects. Therefore, the building of the interface can be automated, if we know all the possible values for some particular objects, which is helpful for the development of advanced graphical objects, such as lists or combo boxes, and so on. The domain knowledge of each object can be increased during the design process of new GUI for the legacy system.

- **Objects Knowledge**

  The knowledge of information objects enables better understanding of the task semantics, and is used to make the target interface of **URGenT** understandable and friendly to the end user. Two aspects of the object's knowledge are expected by **URGenT**.

  (1) Semantics of Objects:

  Obviously, it's almost impossible to know objects' semantics simply by studying the recorded *traces*. However, to make the target interface understandable, a label must be attached next to the input object indicating its meaning. Thus, **URGenT** asks the expert user to provide semantics of the information objects that may appear in the target interface. The previous chapter, *task and domain modeling*, have introduced how to ask and retrieve this knowledge and how to represent it.

  (2) Possible Values for Objects:

  To the novice user, it is typically more popular to select from a list of all possible values of each input object, rather than directly typing in the value in the target interface. As a result, collecting the domain knowledge about possible values of interaction objects is needed for building more advanced input type, such as lists or combo boxes. This knowledge can be provided by the expert user and be helpful for

automated GUI development. This knowledge is represented in terms of *Distinction* and *Content*, too. An example of the knowledge about several objects in the "claim report" task is depicted in Table 4.1.

system	screen	location	values
report	*logon*	(3, 27)	(*eleni*, *sorenson*, *lanyan*, *roland*)
report	*inquiry search*	(26, 4)	(*bruce*, *jim*, *brice*)
report	*query menu*	(34, 8)	(23, 45, 27)

Table 4.1: Knowledge of Variables Values

1. **Distinction**: (*system*, *screen*, *location*)

   System, *screen* and *location* are used to identify the object that has the selected object.

   (1) *system* is the name of the system in which the objects appear, such as the example of the thesis happens in the "report" system.

   (2) *screen* is the unique id of the legacy screen on which the object appears, such as the user name appears on the "logon" screen.

   (3) *location* is the location in the *Legacy Interface* where the object appears, such as the user name has the coordination (3, 27) on the screen.

2. **Content**: (*values*)

   value is the list of all the possible values for the object, such as "eleni", "sorenson", "lanyan", and "roland" are all the possible user names in the task.

- **Design Knowledge**

  Once **URGenT** builds the *Conceptual Interface Model*, it should decide what graphical objects are selected to represent input objects in the target interface. Although based on standard design rules, **URGenT** can automatically generate GUI from the *Conceptual Interface Model*, it also enables the user participation in selecting expected graphical objects to decide the look and feel of the graphical interface.

  To different user types, the selected graphical object for each variable may be dramatically different. For instance, the expert user is quite familiar with the system and the task she is performing, it may be more popular for her to input the value directly instead of choosing it from the list. Alternatively, the novice user doesn't know the system and the task very well, so she prefers selecting from the list that contains all the possible values of each variable, rather than typing the values into the system. This selection knowledge is represented in terms of *Distinction* and *Content*. An example of the knowledge about several objects in the "claim report" is depicted in Table 4.2. Note that, the value "–" indicates the relevant data is either unavailable or unnecessary.

system	screen	location	object	user type
report	*logon*	(3, 27)	*combo box*	*novice user*
report	*inquiry search*	(26, 4)	*text field*	*expert user*
report	*query menu*	(34, 8)	*scroll list*	*novice user*

Table 4.2: Knowledge of Objects Selection

1. **Distinction**: (*system*, *screen*, *location*, *user type*)

   System, *screen*, *location* and *user type* are used to identify the selected object for the particular data.

   (1) *system* is the name of the system in which the objects appear, such as the example of the thesis happens in the "report" system.

   (2) *screen* is the unique id of the legacy screen on which the object appears, such as the user name appears on the "logon" screen.

   (3) *location* is the location in the *Legacy Interface* where the object appears, such as the user name has the coordination (3, 27) on the screen.

   (4) *user type* is the type of the user that prefers this graphical objects selection, such as "novice user" or "expert user".

2. **Content**: (*object*)

   value is the graphical objects selected for the data, which will be used in the GUI design by **URGenT**, such as the "combo box" used for the user name input.

4.2.4 Run-time Execution

This section introduces how to integrate the developed GUI with the legacy system to make it actually executable. As described in the chapter "System Architecture", in order to perform the task, the GUI developed by **URGenT** manipulates the system through the existing interface, with the assistance of the middleware. **URGenT** implements the executable GUI automatically in Java, following the rules described in the previous sections. It builds a run-time component that interprets the interface model, arranges location of objects on screens, and specifies how the objects send or respond to messages. Then the run-time tool actually manipulates the interface, according to the description of dynamic behavior in the interface model. The advantages of building the run-time tool by using Java, an object-oriented and event-driven programming language, are the following.

- Ease of the Interface Generation

  Inheritance feature in Java allows the programmer to add new objects very easily, and to change one attribute of all the interactive objects lower in the hierarchy by making a single change to the common superclass definition. For example, if one more button is needed on the screen, a single line of code can be added to create another instance of the button class, and filling in the details unique to that button, such as the name

of the button, where is it located on the screen, and what action will happen when the button is clicked. Also, the existing interface management library in Java makes it easy to place objects in accordance with the layout rules automatically.

- Platform Independent

  The platform independent feature of Java makes it possible to place the developed GUI on the web, and makes the legacy system world-accessible. Moreover, the swing components in Java ensure the consistency of the look-and-feel of the interface and its objects, so that we don't need to worry that unexpected appearance of the interface may come out due to different environment.

- Extensibility to future interactive system

  Since Java is event-driven programming language, it provides a simpler and more natural paradigm to the interface implementation. The objects communicate with each other by invoking events, which is like sending message by a keystroke or a mouse movement. The event-driven approach is especially suitable for the interactive system, which has strong binding between user action and application event. Therefore, using Java can ensure the extensibility of the software that designs GUI for the interactive system in the future.

At runtime, objects are manipulated according to their types, as introduced below.

- To *Internal* Objects

  During the implementation of the interface, the user never needs to provide the value of *internal* objects. **URGenT** automatically searches their value from knowledge base and sends them to the host system underlying the target GUI, without asking any input from the user.

- To *External* Objects

  The new interface only shows the *external* objects from the *Conceptual Interface Model*. And it asks for each *input* in the *Conceptual Interface Model* only once. To the *inputs* that have multiple numbers in their *sequence id*, i.e. *recurrent data*, **URGenT** buffers them appropriately to deliver them to all screens that use them. To the *intermediate inputs*, i.e. *derived data*, **URGenT** retrieves their values from their source information, i.e. a *display*, and feeds them to the appropriate screens.

- To *Display* Data

  Finally all the *display* data are obtained during the task interaction. For the example in "claim report" discussed in this thesis, **URGenT** automatically develops the final report composed of the display data, and also provides users with alternative to develop report themselves with the retrieved information.

In **URGenT**, although the developed GUI acts like a real interface of the legacy system, the code is not integrated with the system code, but manipulates the system through the legacy interface. Several advantages for the approach are listed as below.

- By separating the newly developed interface from the real system program, the user interface can be changed without dictating the system semantics. This method provides flexibility, portability and evolutionary advantages to the system.
- In **URGenT**, the user can select and customize the graphical objects in the interface, such as menus, combo boxes, buttons, or text fields, without affecting the real system implementation.
- **URGenT** can avoid the effort to extract the interface code and integrate the new interface code with the system code, by keeping the exist interface code untouched, and using a run-time tool implements the dynamic behavior of the interface through the legacy interface.

Chapter 5

Multiple System Interaction

In consequence of the continuous emergence of new technologies, it becomes almost inevitable for the legacy system to ask for some necessary and timely technology upgrades and functional extensions [10]. As a result, it is not enough for *Re-engineering* software to just support or replicate work that currently exists in the system, but it should satisfy specific requirements for functional enhancement [33]. One way to do that is to integrate the legacy system logically with other application software such as web server, calculator or editor. Since the thesis is focused on the task-centered interface migration, an efficient method of developing a new GUI should not only ensure the functional correctness, but it should also have the ability to extend the function of the existing tasks.

It asks for the support of interapplication communication to extend the current task in the legacy system with functions from other applications. For example, suppose that within the task performance, the legacy system might send some numbers to a calculator, which does some calculation on the numbers and then sends the result to an editor or back to the legacy system. The three systems, which are running at the same time, exchange their data during task performance, and finally accomplish the extended task. If the legacy system is integrated with a calculator and an editor, and communicates with them by exchanging data, the existing tasks in the legacy system can be easily extended their function with calculating and editing. As a result, the legacy system is enabled to have the calculation power of the calculator, and editing power of the editor.

Currently, **URGenT** develops the method of adding calculation or editing ability to existing tasks of the legacy system, without changing the original legacy code. In order to grant the user the freedom of extending the functionality of existing tasks of the legacy system, the thesis addresses several problems as follows:

- How to record the interactions of applications other than the legacy system?
- How to integrate the function from other applications with the existing task, and represent them into one *Semantic Task Model*?

- How to coordinate the extended interactions with the original ones, and develop an interface that can make them work together?

5.1 Multiple System Recording

The answer to the first question is to build a monitor around the other application, and record the objects that carry information. In **URGenT**, the following steps are used for the purpose of extending function of the existing tasks.

(a) **URGenT** asks the user to perform the extended task by interacting with the legacy system and other applications, which can accomplish the functionality that is not included in the existing task.

(b) While the performance with the legacy system is recorded by the middleware, **URGenT** records the additional interactions with other applications.

(c) **URGenT** extends the existing task model and represents it in a well-defined model, by first merging the separately recorded interactions into one integral trace.

URGenT records the extended performance by adopting the same terms used in the *Semantic Task Model*, in order to integrate it with the existing task. Therefore, regardless of the variety of applications, the extended task performance are recorded in terms of their *Distinction*, *Temporal Relations* and *Content*:

- **Distinction**: (*system*, *target system*, *task name*)

  (1) *system* is the name of the system in which we are going to extend task function. In the example of task "claim report", the system is named as "report" system.

  (2) *target system* indicates the name of the system with which the legacy system is getting integrated. Currently **URGenT** integrates with the simple desktop tool, such as "calculator" or "editor", that adds some basic desktop function to the legacy system.

  (3) *Task name* defines the name for the extended task generated by integrating the legacy system with other applications, e.g. the "claim report" task.

- **Temporal Relations**: (*sequence id*, *order id*)

  These attributes are used for the sequential information of the extended function.

  (1) *sequence id* is the same as the *sequence id* in the *Semantic Task Model* of the existing task, which indicates the temporal relations between the extended function and the existing task.

  (2) *order id* indicates the sequential relations between objects that happen in the other application or across several applications that are integrated with the legacy system.

- **Content**: (*classification*, *value*)

  These attributes are used to ensure the real execution of the legacy system with other

applications.

(1) *classification* is the types of the different objects in the other applications. They may be quite different, according to different applications. For example, the "operand" and the "operator" are two different data objects in the calculator, while the editor doesn't have "operand" object. However, since **URGenT** assumes that data exchange is the only intercommunication method, two common interactive objects, "copy" and "paste", are defined, which indicate the data exchange happens on the data objects in applications. *Copy* is used for describing highlight action, while *paste* is used for sending information between applications.

(2) *value* is the data exchanged during the interaction. For example, "+" is one possible *value* when the object is "operator" in the calculator system.

For example, suppose during the performance of the "claim report" task, the expert user wants to add calculation and editing function. To accomplish this objective, she

(1) performs the "accident report" task in the "report" system until she navigates into "accident" screen,

(2) highlights a number from the coordination of (5, 4) to (6, 19) on the screen, which is the action item with the sequence id 18 in this task,

(3) copies the number, and pastes it to the calculator, then

(4) in the calculator system, she presses "*", and "3", and "+", then goes back to the "report" system,

(5) continues the "accident report" task in the "report" system until she navigates into "expense summary" screen,

(6) copies the number from (8, 9) to (9, 22) on the screen, and pastes it on the calculator,

(7) in the calculator system, she calculates the result by click the operator "=" on the calculator, and copies this result,

(8) in the editor system, she pastes it on screen at the location of (12, 7).

The recording of the additional interaction is showed in the Table 5.1.

system	target system	task name	sequence	order	classification	value
report	*calculator*	*claimreport*	18	1	*paste*	*none*
report	*calculator*	*claimreport*	18	2	*operator*	*
report	*calculator*	*claimreport*	18	3	*operand*	3
report	*calculator*	*claimreport*	18	4	*operator*	+
report	*calculator*	*claimreport*	26	5	*paste*	*none*
report	*calculator*	*claimreport*	26	6	*operator*	=
report	*calculator*	*claimreport*	26	7	*copy*	467885
report	*editor*	*claimreport*	26	8	*paste*	(12,7)

Table 5.1: Extended Function Recorded in **URGenT**

5.2 Interaction Annotation

In order to describe the task extended by additional functions precisely in the task model, **URGenT** develops a method of integrating the existing task description, i.e. *Semantic Task Model*, with the recording of the additional interactions. Since the *sequence id* recorded for the extended function is the same as the one defined in *Semantic Task Model*, with respect to their temporal relations, the extended interaction can be merged with the *Semantic Task Model* for the functional extension. It is assumed that the "copy" and "paste", i.e. the objects for data exchange, can accomplish the intercommunication between the legacy system and other applications. As a result, two additional interaction objects, i.e. "ask-copy" and "derived-paste" are added into the *Semantic Task Model*.

- "ask-copy"

  In the traces recorded in **Recorder**, since the "copy" action item of the legacy system will only be effective when some area on screen have been highlighted. Therefore, once the "copy" action is invoked in the legacy system, **URGenT** search for the last "ask" action item that is performed previously in the *Legacy Interface*, and change its "ask" type into "ask-copy" type, which can be either *standard-ask-copy* or *selective-ask-copy*.

- "derived-paste"

  If the "paste" action item is invoked in the legacy system, it should happen as one of the "tell" objects in the legacy or other applications. Since this data is derived from the paste panel, it is a derived data. As a result, this object have the classification either as *derived-paste* or *recurrent-derived-paste*.

The integration of functions from different applications results in the *Extended Trace*, which is generated by adding the two additional object types and attaching extended functions to the *Semantic Task Model*. Comparing with the existing *Semantic Task Model*, the little difference in the *Extended Trace* is depicted in Table 5.2. The analyzed result of the "claim report" in the "report" system is the same as shown in Table 5.1, only changed the classification of the data that is associated with "copy" or "paste" action item, for instance, variable 14 and 18 in example "claim report" task.

Sequence	Value	1st Phase	2nd Phase	3rd Phase
18	$(5,4)to(6,19)$	$ask,\ var14$	$problem-specific$	$standard-ask-copy$
26	$(8,9)to(9,22)$	$ask,\ var18$	$problem-specific$	$standard-ask-copy$

Table 5.2: Change of The Analyzed Trace After Functional Extension in **URGenT**

5.3 Run-time Execution

During the development and deployment of the target GUI from the *Conceptual Interface Model*, several rules for the run-time execution are added. The legacy system and other applications are working concurrently during execution, until they need to exchange data by copying or pasting. For the example, in "claim report" task, when the "report" system goes to action sequence 18, it copies the highlighting information to other application, e.g. the calculator. Therefore, **URGenT** turns to operate on the calculator, performing all the interactions with the same sequence id, e.g. sequence 18, but with increasing order id from one to the highest. Meanwhile, interactions in the legacy system are executed concurrently, unless the interactions cannot continue without further getting (giving) some data from (to) other applications. After the operation with the same sequence id in other applications is finished, **URGenT** continues with next recorded sequence id. If there is no next sequence id left for other applications, **URGenT** finishes the execution in the legacy system until tasks are accomplished.

After identifying requirements of the extended task, **URGenT** generates *Semantic Task Model* out of *Extended Trace*, and maps the *Semantic Task Model* into a *Conceptual Interface Model*, which can support the generation of the GUI to achieve the extended task. **URGenT** assumes that the other application is a simple desktop tool in which the interactions can be achieved by the defined operation on fixed data that are "told" by the legacy system. Based on this assumption, the interactions for the extended task can be carried on without any input from the user. Therefore, interaction objects in other applications are all treated as *internal objects* in the *Conceptual Interface Model*, in order to support the execution of the extended function in the target GUI, as well as avoid affecting the layout of the new GUI. The specification of those additional *internal objects* is depicted in Figure 5.1.

As a result, although **URGenT** can extend the task of the legacy system with other applications of simple desktop tools, such as calculating or editing, it currently generates the target GUI that hides the extended functions from the user, and performs them automatically without asking users. In other words, for the extended task, the currently developed interface is the same as the one developed before functional extension, and **URGenT** performs all the extended interaction underlying the new GUI.

system :	report
target system:	calculator
task name:	claim report
sequence+order id:	18+1
classification:	paste
value:	none
system :	report
target system:	calculator
task name:	claim report
sequence+order id:	18+2
classification:	operator
value:	*
system :	report
target system:	calculator
task name:	claim report
sequence+order id:	18+3
classification:	operand
value:	3
system :	report
target system:	calculator
task name:	claim report
sequence+order id:	18+4
classification:	operator
value:	+
system :	report
target system:	calculator
task name:	claim report
sequence+order id:	26+5
classification:	paste
value:	none
system :	report
target system:	calculator
task name:	claim report
sequence+order id:	26+6
classification:	operator
value:	=
system :	report
target system:	calculator
task name:	claim report
sequence+order id:	26+7
classification:	copy
value:	467885
system :	report
target system:	editor
task name:	claim report
sequence+order id:	26+8
classification:	paste
value:	(12,7)

Figure 5.1: Specification of Extended Interactions

Chapter 6

Related Work

This section introduces several research projects with different perspectives in this area, and compares the work in the thesis with those researches. The related researches are classified into two groups, according to the two major processes in *Interface Migration*, *system understanding* and *interface generation*.

- *system understanding* [12] understands the system functionality and constructs an abstract model, which describes the requirements for the the existing system interface.
- *interface generation* [29] generates the user interface, based on the requirement description from the abstract model that is generated by the *system understanding*.

Since *system understanding* is a quite time-consuming process, it could be a big contribution to automate the understanding process as much as possible. This thesis focuses on the introduction of the researches that understand the functional requirements for the system interface. Those researches in this area are searching for better method to understand the interface functionality, and most of them use the method like extracting the interface subcode from the system code and analyzing it, based on the knowledge of the programming language and the system domain. The section of *Program Understanding* introduces one typical method in this area and compares it with the trace-understanding method used by the thesis.

For *interface generation*, a lot of development effort is currently used to re-designs the interface so that it can meet the requirements of new environment or new technology. Model-based *Interface Migration* can greatly automate this process, and ensure the functional correctness as well as enhance the performance of the target interface [32]. Therefore, the section of *Model-based Interface Design* introduces several projects in the area, and compares them with ours.

6.1 Program Understanding

Research in *Reverse Engineering* consists many different approaches, such as formal transformations, pattern recognition, and reuse-oriented approaches [17]. Since in many cases, there is no documentation of design and no maintenance history available for a legacy system, the source code seems to be the only written definition of the system. As a result, regardless of the approach, researchers typically use programming knowledge, domain knowledge, and comprehension strategies to form an understanding of the program and reconstruct them.

In the chapter of *Introduction*, the thesis has already introduced a domain-based approach for program understanding, **Synchronized Refinement** [35]. In this section, another domain-based program understanding project, **Rigi** [17], is discussed with more detail of the code analysis method. **Rigi** supports the understanding of the existing system by analyzing the code and graphically describing the system functionality. The method of code understanding in **Rigi** involves two phases: *identification* and *discovery*.

1. The *identification* phase is where the system's current components and their dependencies are identified [17]. It involves code analysis integrated with the syntactic rules of the program, which results in a graph representation where nodes represent system components (such as types, variables, modules, etc.) and arcs represent relationships among the nodes (such as 'depends on', 'used by', 'contains', etc.).

2. The *discovery* phase is a complex interactive activity. It is concerned with the extraction of design abstractions for the system, which is to understand the functionality of the system components, and represent the system components in abstract forms [7]. It addresses the issue of gaining a general understanding of the system prior to specific evolution activity. The user may build up hierarchical subsystem components which optimize software engineering principles such as low coupling and high cohesion [17]. *discovery* can also include the reconstruction of design and requirements specifications (often referred to as the "domain model") and the correlation of this model to the code, which makes it possible to extend functionality for the system, if necessary. Finally, this phase generates system specification which defines requirements and behavior for the software components of the system [7].

As mentioned in the first chapter, the understanding method by code relies heavily on the assumption of the program structure. Therefore, when the program is poorly structured, changed substantially by several designer, or even not available, the method may be too expensive or even infeasible. In the contrary, the trace-understanding method used by the thesis is always feasible, as long as the system is functioning correctly with no exception in the task performance.

6.2 Model-based Interface Design

In the *Interface Migration* process, the *Reverse Engineering* phase is typically followed by a forward engineering phase that moves from the high-level abstraction to low-level physical implementation and reconstitutes the *Legacy Interface* into a new form which is accustomed to the new environment and technologies. Most of the current approaches are using a traditional user interface management system (UIMS) to develop interfaces rapidly. The interface designer is asked to create the interface by dragging and placing interface objects on the screen [10]. However, this approach distracts the designers from the design of optimization for task performance, but put on the designer the unnecessary burden, such as understanding the object semantics, studying layout rules and applying selection rules for graphical interaction objects. Furthermore, the UIMS only provides limited support for specifying interaction of the GUI with the underlying system, and does not incorporate the support for run-time dynamics, i.e. the control of the objects [32].

Most of current research on interface modeling describes an interface as a hierarchy of objects. Each of the objects is described in terms of a set of pre- and post-conditions [32]. Pre-conditions of objects determine their visibility and enabled or disabled status, and postconditions of objects are associated with functionally different actions performed on objects [10]. They are used together with actions in the User Interface Design Environment (UIDE) to describe partial semantics of the applications. A set of interface objects with preconditions and postconditions can generate a complete dialog implementation, and this technology can bring several additional features to the interface design, such as context-based help generation.

Model-based user interface design is centered around a description of application objects and operations at a level of abstraction higher than that of code. An efficient model can be used to

(a) support multiple interfaces for different platforms,

(b) help separation of the interface and the application, which describes input sequencing in a simple way,

(c) ensure consistency and completeness of the interface, and support the generation of context-specific help and the design of the interface.

There are two kinds of task model:

- Formal Grammar Description

  Task Action Grammar (TAG) and Task Analysis for Knowledge Description (TAKD) [35] represent the task using the formal grammar description that is familiar and understandable to computer system.

  (a) TAG generates Backus-Naur Form (BNF) to represent task structure, however,

TAG is unfortunately only suitable for application to low level user interface features. Furthermore, BNF is no longer a popular representation for system design work [25]. (b) TAKD is capable of generating formal output, and is extended to allow it to produce analysis outputs in formats familiar to computer system designers. TAKD produces task analysis outputs using several main stages:

1. Activity List Construction:
   An Activity List (AL) is a general description of a set of tasks. There are no constraints on the requirements capture method used to generate an AL. For instance, task-orientated interviews or task scenarios (i.e. idealized task descriptions) can be the source of an AL.
2. Specific Action and Specific Object Selection:
   Virtually all task analysis methods have some concepts of actions and objects. In TAKD the Specific Actions (SAs) and Specific Objects (SOs) are the lowest level of data that will be subjected to analysis. SAs and SOs are selected directly from the AL and the iteration between editing the AL and selecting SAs and SOs improves the consistency of the AL.
3. Task Descriptive Hierarchy Construction:
   A Task Descriptive Hierarchy (TDH) is a highly specialized hierarchy. In TAKD, task analysts concentrate their design skill at understanding the tasks they are analysing during its construction.
4. Knowledge and Sequence Representation:
   The AL can be rewritten in TAKD's standard notation so that the whole task can be viewed at any level of abstractness. TAKD's standard notation consists of sets of Knowledge Representation Grammar (KRG) sentences that describe the single traversal from terminal leaf node to top node in AL. The task sequence from the KRG sentences is represented as Sequence Representation Grammar (SRG) lists, which separates parallel, interrupted or shared tasks by constructing blocks of related AL lines.

- General Knowledge Structure:
  Task Knowledge Structure (TKS) represents the hierarchy goal structure of the task. It assumes that the user knowledge to the task has already be captured. The LOTOS [30] formal specification technique is used for the representation of the TKS model. The major components of the representation are:
  (a) goals that includes temporal information,
  (b) procedure that consists of actions,
  (c) actions that are performed on objects,

(d) objects in terms of attributes and relationships to other objects.

They are represented in two types of knowledge structures:

(1) taxonomic knowledge that is composed of objects, attributes and actions on objects, and

(2) goal-oriented substructure that represents the goal, subgoals and their enabling state which implies the sequential constraint and dependency.

The remaining of this section will introduce four projects which focuses on the model-based interface design. They are **ADEPT**, **TRIDENT**, **Mecano** and **Mastermind**. **ADEPT** and **TRIDENT** projects concentrate on the task-based interface design in modeling from the system documentation or the user knowledge of the task. **Mecano** and **Mastermind** emphasize on the GUI development from existing models, which is generic to all tasks in the system.

6.2.1 Adept

The **Adept**(Advanced Design Environment for Prototyping with Task model) [34] is a project to support the user interface design based on the model, which represents the user's existing knowledge about the task performed on the user interface. It uses the extended version of TKS model, which contains a goal sub-structure component to describe user's goal, and a procedural component to describe the action sequence of the goal. These components in TKS model can feed forward into a design, and **Adept** lets the user participate the design in every step.

ADEPT generates both a user model and a task model, integrates two models, and develops the model of proposed task from knowledge to design. Part of the design in **ADEPT** is expected to take place at the task level. **ADEPT** is also concerned with supporting the designer's participation in the design process. For instance, it has task model editor that is proved to be a valuable way of obtaining user feedback and getting users directly involved in modifying the design representation. It also enable the user to define the user knowledge by using a user model. After the user participation in the design, an abstract model of the new interface can be derived from the designed task model by producing a hierarchy of abstract interaction objects that is similar to the TKS goal structure.

Although **Adept** deploys the formal TKS goal structure to construct the task model, since there is no formal method for the design of the interface based on the TKS goal structure, it cannot totally automate the process of interface design from the task model.

6.2.2 TRIDENT

Another similar project, **TRIDENT** (Tools foR an Interactive Development EnvironmeNT) [14], is also a model-based approach for the user interface design of highly interactive sys-

tems. It starts from a task analysis of the future application to a visible and working prototype of the presentation. **TRIDENT** generates two models:
(a) *presentation* forms the static visual layout of the screen made up of interaction objects.
(b) *conversation* is responsible for the dynamic behavior of the interaction objects that are manipulated by the user.

Their constructions rely mostly on a task model with operational requirements and an extended *entity-relationship-attribute model*(ERA) that describes functional requirements. In order to help analyzing the task and functional requirement, and selects appropriate interaction style for the task, the task model is integrated with the ERA model, and graphically depicted by the *activity chaining graph* (ACG). An *activity chaining graph* (ACG) is a graph of information flow between the application domain functions which are necessary to perform task goal. It provides a functional invariant for the interactive task, independent of user characteristic.

To construct the *presentation* model, the presentation unit(PU) is extracted from ACG, which is defined for each sub-task in each interactive task. It can be in physical level which decides objects selection in ACG, or in logical level that abstract both presentation of windows and behavior on each object in windows.

To construct the *conversation* model, the approach is to build a hierarchical object-oriented architecture of dialog objects from the task model in ACG. Each dialog object is responsible for one piece of the global conversation. Its behavior can be partially represented graphically by a state-transition diagrams.

Without sufficient knowledge to back up, **TRIDENT** process is not highly automated, such as the PU divided from the ACG that must have the user participation. Moreover, since it does not generate explicit dialog model, the prototype it develops contains only the static simulation of the designed interface without the dynamic behavior, i.e. it doesn't support the real execution of the interface.

6.2.3 Mecano

The chapter of *System Architecture* presents a generic framework that automates interface generation environments employing data models. **Mecano** [4] uses domain model instead of data models with the same framework to generate interfaces. It builds a model-based interface development environment that generates both the static layout and the dynamic behavior of UI. Furthermore, it integrates an interface design tool with an inference engine that uses style rules for selecting and placing GUI controls, and allows a single data model to be mapped onto multiple GUI's by substituting the appropriate rule set. This methodology represents a first step toward a GUI-independent run-time layout facility.

The domain model is a high-level knowledge representation, which captures all the def-

initions and relationships of the system domain. In **Mecano**, it is constructed using a frame-based representation language that defines class hierarchies. It contains two important relationships:

- *is-a* relationship defines the class hierarchy and is used by the intelligent designer to specify the interface-navigation schema among objects.
- *part-of* relationship is used to decide the objects grouping into windows.

As a result, by using a domain model as input, an intelligent designer can develop both the dynamic dialog specification and the static layout of the interface, and stores them in an interface model, which can be transformed into UIMS specification. Since this interface model contains all facets of an interface design including interface objects, presentation, dialog and behavior, it facilitates the generation of the interface, which can be implemented by a run-time system.

How to collect enough information in the domain model in order to support the design process is a big challenge for **Mecano**. The method is to collect the knowledge increasingly instead of from scratch, and build a domain model editor for user participation to improve its content.

6.2.4 MasterMind

MasterMind [36] is a project for integrating several models to generate interfaces, which fits for the rich and powerful interactive applications. It supports the automatic generation of user interfaces from several declarative models. Three types of models are supported in this project: the *Presentation Model*, *the Dialog Model*, and the *Interaction Model*.

- The *Presentation Model* (PM) describes the objects appeared on the end user's display and the dependencies among objects.
- The *Dialog Model* (DM) describes the various low-level input activities that may be performed by the end user during the interaction with the generated GUI, as well as the relative orderings of the activities and the resulting effects they may have on the system.
- An *Interaction Model* (IM) specifies the set of possible low-level interactions between the end user and the run-time environment. It is determined by the underlying toolkit technology upon which **MasterMind** code is generated.

Beside the models, there are several engines used to accomplish the design in **MasterMind**. (1) *Application Wrapper* (AW) specifies the interface between **MasterMind** and application functionality. its specification includes method and data item declarations for features

that the application makes accessible to the user interface.

(2) *Dialog Notation* (MDL) allows designers to specify the permissible ordering of the end-user interactions that carry out some task. The notation also provides operators for expressing preemption.

To go from models to executable code, **MasterMind** employs model-specific code generators. Because there are necessary dependencies among models, model instances must refer to elements of other model instances. By focusing attention on a single aspect of a user interface, a model can be expressed in a highly-specialized notation. Model-specific compilers generate modules of code from each model, and these resulting modules are composed.

A distinguishing characteristic of **MasterMind** is that the model-specific code generators work independently of one another. However, inevitably, functionality described in one model will overlap with or is dependent upon functionality described in another. So the redundancy and dependency of the models have to be considered and appropriately handled during the code generation from multiple models. This is a challenge to the model composition.

Chapter 7

Conclusions And Future Work

This thesis proposes a technique of task-centered interface migration, which focuses on studying recorded task traces to learn the task semantics and automating the graphical interface development for these tasks. Chapter one discusses the importance and current status of the research in this area. Chapter two describes the system architecture for implementing this technique. Chapters three through five present the detail of its implementation in different processes, and its methods of extending the task functionality. Chapter six introduces several related researches and compares them with this technique. **URGenT** is the prototype system developed for task-centered *Interface Migration*. In the context of user participation, the overall **URGenT** process consists of the following steps:

Expert Users: perform the actual task several times in the standard way, so that **Recorder** can retrieve the task navigation plan and data flow.

URGenT: analyzes the traces recorded by **Recorder**, and generates a description of the task in a well-defined format.

Expert Users: review the task description, and verify it by confirming analysis result or providing semantics for each information object.

URGenT: develops the *Conceptual Interface Model* by mapping from the *Semantic Task Model*.

Expert Users: review the *Conceptual Interface Model*, and modify it or add more information for the interface design.

URGenT: integrates the *Conceptual Interface Model* with certain design rules, and generates the GUI that can actually perform the task with the legacy system.

The most interesting features of the **URGenT** prototype system are:

- *Trace-Based Analysis*

  URGenT learns the task semantics by analyzing the recorded traces of the task performance. Therefore, this method does not have to study the lengthy, complex,

and glued code of the legacy system.

- *Interactive Domain Modeling*
  URGenT requires the user's participation in order to collect domain knowledge. This knowledge will be used to make the labels of the widgets more meaningful and thus the interface easier to learn and to use.

- *Optimization and Extension*:
  URGenT develops the target GUI by studying the logic of the task as well as taking advantage of the GUI feature, therefore the target interface has the potential of optimizing the user-system interaction. Meanwhile, **URGenT** is able to extend certain system task by adding more standard functions into the target interface.

- *Model-Based Interface Generation*:
  URGenT maps the *Semantic Task Model* to the *Conceptual Interface Model*, a platform independent description of interfaces, and develops the actual GUI based on the *Conceptual Interface Model*. Therefore, **URGenT** accomplishes the *Interface Migration* based on a higher level of system understanding, rather than just transforming objects between different platforms.

In the following sections, the technique implemented by **URGenT** is evaluated, then its contribution and advantages over other methods are discussed, and finally some perspectives on potential future work are presented.

7.1 Evaluation

An approach to evaluate a theory is to develop a prototype system that implements the theory, and predict the time and effort taken to carry out the idea with the system, as well as measure the quality of the generated products [1]. **URGenT** is the prototype system developed by the thesis. It implements the research idea of the thesis, generates GUI to wrap aspects of the existing interface, therefore **URGenT** demonstrates the feasibility of the interaction-based approach to both the users and the designers [14]. This section will evaluate **URGenT** by developing GUIs for several test examples, and comparing them to the existing interfaces in the legacy system.

URGenT is the prototype system that is manipulable, and can be adjusted. As for its efficiency, the process of *Interface Migration* is semi-automatic, and certain functions, such as annotating domain knowledge with the task description, are included to save time and cost in analyzing task and generating GUI. It may make the cost of interface migration lower than that of rewriting interface code from scratch.

The generality of **URGenT** is measured with respect to the class of systems it applies to and the class of tasks it addresses. Typically, most processes in information systems are oriented towards data manipulation. Therefore, they usually share several basic tasks, such as creation, deletion, query, and modification of data and data records [26]. In the thesis, **URGenT** focuses on the *Interface Migration* for data-oriented tasks of the information domain, and develops the system interface that often closely corresponds to those tasks. As a result, a method of *Interface Migration* based on data interaction might be widely applicable to tasks in the information systems.

In order to evaluate **URGenT**'s interface migration process, let us compare the quality of the GUIs developed by **URGenT** with that of the legacy interfaces. The following criteria are used in the comparison of interfaces.

- *Correctness*

  Does developed GUI have the same functions as the legacy interface?

- *Efficiency*

  Is the time and workload that users put into the task performance with the GUI less than the user has to do on the legacy interface?

- *Usability*

  How friendly is the new GUI to users, i.e. how easy it is to understand and to use?

For the purpose of evaluating the system, several tasks within two different systems have been conducted with **URGenT**. Those system tasks are designed to investigate the GUI effects on the efficiency of its process. Several of them are operated in an off-line system, which maps the screen snap-shots of an insurance company into a file. The users interact with the file to emulate the real interaction with the system. The other tasks are executed in an on-line system, such as a library system on the web. This section will evaluate the **URGenT** system by describing a "claim report" task, which has been discussed across the previous chapters, in an off-line insurance system.

- Correctness

  The developed GUI acts as the front-end of the legacy system. Currently **URGenT** uses a legacy interface as the bridge to exchange data between the GUI and the system, so that the GUI can manipulate the system to accomplish the task. The **URGenT** system is built based on the assumption that the task is deterministic, which means the recorded task traces contains the same actions with no exception. When the input contains recorded traces of non-determinism task, the GUI will fail to be correct for this task. However, for deterministic tasks, e.g. "claim report" task, the GUI generated by **URGenT** accomplishes the task in the similar way as what the existing

interface does. Although interacting through the legacy interface, i.e. feeding personal data to the system through the legacy interface, or retrieving the data provided by the system from the legacy interface, slows down the GUI performance, the speed is not the major concern here, because as mentioned before, **URGenT** is a prototype system that is constructed with limited function and performance, and only used for evaluation purpose.

- User Efficiency

  When the user performs the "claim report" task with the the *Legacy Interface*, s/he has to enter her personal data, e.g. user name, twice into different subsystems, write down the data provided by the system screen, such as the claim number, then enter the data into three different screens in order to obtain relevant data to build the final report. In comparison, since the new GUI delivers (or retrieves) the static information of the task into (or from) the system without asking the user, the user only has to enter the personal data and the provided data once, and consequently get relevant data from different screens with the help from **URGenT**. Therefore, the newly developed GUI can avoid the iteration of the same operation. Furthermore, the task-centered design of the GUI, which tailors the GUI in the context of each task, may optimize the interaction between users and the system. In conclusion, an evolved **URGenT** could save both time and cost for its GUI implementation and make it more efficient.

- Usability

  A target GUI is developed to accomplish each particular task, and GUI has special objects, such as the help dialog and the exclusive radio button, that help users to understand tasks and relations between exchanged data. For example, Figure 4.2 depicts the GUI of "claim report" task, which has user names listed in a combo box, and client names shown in radio selection group. To the new users who are not very familiar with the system tasks, the GUI developed by **URGenT** is theoretically easier to use. Therefore, migrating text-based generic interface to task-centered GUI improves the usability of the system. However, this advantage is not applicable to users accustomed to the current legacy interface, who prefer fast performance to more hint and information of how to accomplish tasks.

7.2 Limitations and Future Work

URGenT is the prototype system that is used as a test bed for further development and experiment of the thesis research. It is currently in its infancy and far away from a complete working system. As mentioned in previous chapters, the first prototype of the **URGenT** system has made explicit simplification on the proposed architecture in several aspects.

For example, it cannot handle the condition of task execution with exceptions, and has no advanced GUI design involved. Moreover, it is not able to integrate two legacy system for more complicate tasks. Therefore, more research work and implementation steps have to be carried out to improve **URGenT**. Some of its limitation and the possible involved future work are introduced as follows.

7.2.1 Non-Deterministic Task Analysis

Although in order to simplify the task analysis, **URGenT** assumes the deterministic task performances, i.e. all the performance for the same task has the same action sequence and screen navigation, the task performances in the real world are normally non-deterministic. For example, information retrieval is the task which is driven by situational triggers (e.g. intermediate search result) [18], and should take account of both "planned" and "situated" aspects of the task. As a more specific example, the interface should stop interaction with the system. When it meets an error message during task performance. Delicate methods of handling the non-deterministic task in the GUI should be worked on in the future. Current thought of the first step towards this end is to use a graceful degradation for unexpected behavior, i.e. picking up the non-deterministic task performances and putting them into special cases for the task. Because the task for information retrieval is the high-level task which often exhibit procedural dependence among the low-level interactive tasks, and the procedural dependence can be deployed for inferring the task goal and navigation plan for this kind of tasks, the non-deterministic traces can actually help understanding the task semantics further. Probably the dynamic user interface will be used in this direction of work.

7.2.2 User-Support Tools

The second aspect to be enhanced is the support tools developed for both the system user and the end user. More functionality and flexibility should be provided to the user, which makes **URGenT** easier to understand and use. For user participation in the understanding process, a better format should be developed for the task viewer. In the future, **URGenT** should represent the parallel traversals in the task performance that can be used to represent the extended function, as well as the existing task. It should show more information to users about the relation between subtasks of the task, and arrange them in a way that is easy to see and understand. The user model [14] is the user stereotype that abstracts a particular category of users in the population: the same task can be accomplished in very different ways by different users according to their habits and skills. In the future, **URGenT** should have the user profile that provides guidance in discriminating between alternative interface design which satisfy the interface model. Also, there's some user-oriented data that should

be asked to be entered by the user every time, for the sake of security, such as the password in the "accident report" task. Therefore, **URGenT** should treat them as normal variables, and not store them in the *Domain Knowledge Model*. All this information needs more user participation in the development process. As a result, **URGenT** should develop appropriate tools to present them to the user and ask for feedback.

7.2.3 Advanced GUI Design

In order to develop more advanced GUI capabilities, the following aspects should be investigated in **URGenT**.

1. Multiple Interface Models

   Related research [32] decompose GUI design into the construction of separate models, each of which is declaratively specified to some single coherent aspect of a user interface, such as its appearance, or how it interacts with the underlying system functionality. This could make the representation of the interface more complete and helpful for future design. Furthermore, modularization and integration of interface models are required in a model-based interface design, when there are multiple designers for one interface. For example, one designer could be responsible for screen layout, one for dialog, and one for application software. Meanwhile, the declarative style of the *conceptual interface model* brings out several beneficial features such as design critics, context-sensitive help, and dynamic reconfiguration [36]. Therefore, **URGenT** in the future may develop multiple interface models that must pay special attention to functionality that overlaps different models.

2. More Selection of Interface Objects

   Objects in the interface model should be further classified into different data types, in order to exploit the more flexible and advanced feature of graphical interface. Also, more knowledge of each object, such as the possible values, could be very helpful to GUI development [22]. For example, integrating the data type of each piece of information with the selection rules can help automate the graphical objects selection in GUI development. There are different object selection for various data types, such as time, date and boolean, integer, real, graphic data, alphanumeric data, data group, etc., and for different information display, such as static data and dynamic data and group display [19]. For each data type, there are different analysis criteria for selection of the graphical objects. For example, the selection rules for the numeric data need to judge the objects' attributes as: data content, limits, range, and precision, while selection rules for enumerated attributes are based on the characteristics such as item set, label and size of the objects [22].

3. More Complex Interface

   To date, this project applies methods for simple report tasks, for which a simple GUI generation process using a two-column layout can generate the interface. However, more complex tasks will require more elaborate approaches to the problems of window organization and object layout. For example, some tasks should customize the interface by splitting multiple objects into two or more windows, according to their logical role in the task [22]. Also the controls appearing in the screen sometimes should be in certain order according to the relation of the data objects they represent. They could be derived by examining the objects they affect and the state of the system before and/or after they are invoked [22]. Meanwhile, the dynamic placement of the graphical objects on screen support participatory layout design involving the user, who in this case can actually place the graphical objects on the screen. In the future, **URGenT** should support the expert user to decide the layout of the GUI, by provide the relations between objects and the GUI-specific control selection rules that decide how to map elements of the data model onto a particular control in the target GUI.

7.3 Contributions

In the thesis, a method of trace-based task analysis and a theory of task-centered GUI design are proposed for *Interface Migration*. In order to evaluate the method and the theory, a prototype system, **URGenT**, was designed, implemented and introduced in the thesis. **URGenT** intends to use an effective modeling process, which builds a strong foundation for subsequent interface development, that should result in less iteration for future design. Therefore, it develops

(1) a technique of collecting knowledge in the task domain, which is used in the *Domain Modeling* process,

(2) a method of understanding the functional requirements of existing tasks, which is the essential element in the *Task Modeling*, and

(3) a framework for constructing representational models, which supports both the *Task Modeling* and the *Domain Modeling*.

The main contributions of the thesis can be summarized as follows:

- *Novel Approach to Interface Migration*

  The thesis develops the novel concept and method of trace-understanding and task-centered interface migration. Based on the recorded traces of the user's dynamic interaction with the system rather than the cumbersome code, **URGenT** extracts an understanding of the user's task, and generates the GUI for the task. Comparing with code-based analysis and generic interface design, this approach has the following

advantages:

(a) Accuracy:

Since the traces record the actual performance of the task, the captured meaning of the task is up-to-date.

(b) Efficiency:

By analyzing the traces, **URGenT** avoids the effort of studying the long and glued code in order to understand the task. Furthermore, since the interface design is task-centered, it is tailored to meet the needs of the particular task.

- *Optimized Interaction*

  The optimization of the developed GUI, means less decision and less iterative operation for the user. **URGenT** simplifies the steps of interactions for each task, and avoids the iterative operation as much as possible. Users only need to convey the information pertinent to the task and dynamic to each performance, and do not have to provide the same information more than once. As we know from the previous introduction, **URGenT** achieves this by using new GUI that only asks the user for the problem-specific variables once, caches them if there's any further use, and sends the constants without asking. By this method, users are enabled to sufficiently use the system without redundant operations.

- *Functional Extensibility*

  URGenT develops a method to integrate different data sources, such as the domain model, traces recorded for each task, and the records of the basic desktop function. Therefore, it is capable of extending the existing task by merging those data flow and interactions together.

- *System Reusability*

  URGenT generates specific views of the system, from various task perspectives. Essentially it wraps the legacy system with well-defined APIs to enable its interaction with other systems. Other applications can use this feature to integrate with the system from this particular view, without the understanding of the whole system.

- *Working Prototype*

  The thesis has experimented and realized the concept and the method in a working system, i.e. **URGenT**. Based on the traces of task performance, **URGenT** classifies the types of the flow information in the system, then specifies the task in a model for the legacy interface. **URGenT** transmits the semantics task model into a conceptual interface model independent of platforms, and finally develops from the interface model the new GUIs, which can really manipulate the system for task accomplish-

ment. Personally I have learned a lot through research and hand-on experience with this implementation, and have had a lot of fun in addition to hard work.

Bibliography

[1] A. Aboulafia, A.H. Jorgensen, and J. Nielsen (1993). *Modelling in User Interface Design: Designers' Views of Application and Requirements.* Proceedings of the 16th IRIS: Information Systems Research Seminar in Scandinavia, pp.298-307.

[2] Andreas Girensohn, Beatrix Zimmermann, Alison Lee, Bart Burns, and Michiael E. Atwood. *Dynamic Forms: An Enhanced Interaction Abstraction Based on Forms.*

[3] Andreas Girgensohn, Alison Lee. *Seamless Integration of Interactive Forms into the Web.*
http://www.fxpal.xerox.com/papers/gir97/

[4] Angel R. Puerta, Henrik Eriksson, John H. Gennari, and Mark A. Musen. *Model-Based Automated Generation of User Interfaces.* http://www-ksl.stanford.edu/KSL_Abstracts/KSL-94-51.html

[5] Bailey, W.A., Knox, S.T, and Lynch, E.F. *Effects of interface design of user productivity.* in Proceedings of the Conference of Human Factors in Computer Systems. New York: ACM, 1988, 207-212.

[6] CEL corporation home page: http://www.celcorp.com.

[7] CDRL Sequence 05507-001D. SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS PROGRAM. *Cleanroom Engineering Specification.*
http://source.asset.com/WSRD/ASSET/A/518/elements/HandbookVol_4.txt.

[8] Clayton Lewis, and John Rieman. *Task-Centered User Interface Design.* Shareware.
http://www.acm.org/ perlman/uidesign.html.

[9] Dan Diaper. *HCI and Requirements Engineering - Integrating HCI and Software Engineering Requirements Analysis.*
http://www.acm.org/sigchi/bulletin/1997.1/diaper.html.

[10] Daniel F. Gieskens and James D. Foley. *Controlling User Interface Objects Through Pre- And Postconditions.* In Proceedings of the CHI'92 Conference, May 3-7, 1992. pp 189-194.

[11] *Domain Based Design Documentation and Component Reuse and their Application to a System Evolution Record.*
http://www.cc.gatech.edu/reverse/dare/final_report/

[12] E. Merlo, P-Y Gagni, J.F. Girard, K. Kontogiannis, L.J. Hendren, P. Panangaden, and R. De Mori. *Reengineering user interfaces.* IEEE Software, v. 12, no. 1, January 1995, pp.64-73.

[13] Eva, M. (1994) *SSADM Version 4: A User's Guide.* Second Edition. McGraw-Hill.

[14] Francois BODART, Anne-Marie HENNEBERT, Jean-Marie LEHEUREUX, Isabelle PROVOT, and Jean VANDERDONCKT *A Model-based Approach to Presentation: A Continuum from Task Analysis to Prototype*, In Proceedings of DSV-IS'94, June 8-10, 1994.

[15] Fraser Hamilton, Peter Johnson and Hilary Johnson. *Task-related principles for user interface design.*
http://www.dcs.qmw.ac.uk/research/hci/.

[16] Gail C. Murphy, David Notkin, and Kevin Sullivan. *Reflexion Model.* http://www.cs.ubc.ca/spider/murphy/papers/rm/fse95.html.

[17] H. A. Mller, K. Wong, and S. R. Tilley. *Understanding software systems using reverse engineering technology.* The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS 1994). http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml.

[18] H. Ulrich Hoppe and Franz Schiele. *Towards Task Models For Embedded Information Retrieval.* In Proceedings of the CHI'92 Conference, May 3-7, 1992. pp 173-180.

[19] Jean Vanderdonckt *A Corpus of Selection Rules for Choosing Interaction Objects.* Technical Report TR 93/3, University of Namur, August 1993.

[20] Maluf David A. and Wiederhold Gio. *Abstraction of Representation for Interoperation.* Foundations of Intelligent Systems, Zbigniew Ras and Andrzej Skowron Eds. Lecture Notes in Artificial Intelligence, subseries of Lecture Notes in Computer Science, Springer Verlag, Vol. 1315, ISBN 3-540-63614-5, 1997. http://www-db.stanford.edu/ maluf/publications.html.

[21] Mark Green. *The University of Alberta User Interface Management System.* Proceeding of SIGGRAPH, 12th Annual Conference, San Francisco, CA, July 1985.

[22] Mark H. Gray, Dennis J.M.J.de Baar, James D. Foley and Kevin Mullet. *Coupling Application Design and User Interface Design.* In Proceedings of the CHI'92 Conference, May 3-7, 1992. pp 259-266.

[23] Martin Frank & Pedro Szekely. *MASTERMIND - Adaptive Forms.* http://www.isi.edu/ mm-proj/

[24] Martin Robert Frank. *Model-Based User Interface Design By Demonstration and By Interview.* Ph.D thesis in Georgia Institute of Technology.

[25] Melody Moore and Spencer Rugaber. *Using Knowledge Representation to Understanding Interactive Systems*, Proceedings of the Fifth International Workshop on Program Comprehension, IEEE Press, May 28-30, Dearborn, Michigan, 1997.

[26] Melody Moore, and Spencer Rugaber. *Issues in User Interface Migration*, Proceedings of the Third Software Engineering Research Forum, Orlando, FL, Nov 10, 1993.

[27] Melody Moore. *Rule-Based Detection for Reengineering User Interfaces.* in Proceedings of the Third Working Conference on Reverse Engineering(WCRE), IEEE Computer Society Press, Monterey, California, November 8-10, 1996.

[28] Melody, Moore. *Representation Issues for Reengineering Interactive Systems.* ACM Computing Surveys, Vol. 28 es, Number 4, December 1996. http://www.acm.org/pubs/articles/journals/surveys/1996-28-4es/a199-moore/a199-moore.html.

[29] M Sanz, and E.J Gomez. *Task Model for Graphical User Interface Development.* Technical Report gbt-hf-95-1, Grupo de Bioingenieriay Telemedicina, Universidad Politecnica de Madrid, March 1995.

[30] Panos Markopoulos, Stathis Gikas. *Towards A Formal Model For Extant Task Knowledge Representation.* http://www.dcs.qmw.ac.uk/ markop/publications.html. ftp://ftp.dcs.qmw.ac.uk/pub/hci/publications/94-MarkopoulosP-1.ps.gz.

[31] Pedro Szekely, Ping Luo and Robert Neches. *Facilitating the exploration of Interface Design Alternatives: The HUMANNOID Model of Interface Design.* In Proceedings of the CHI'92 Conference, May 3-7, 1992. pp 507-515.

[32] R.E. Kurt Stirewalt, Spencer Rugaber. *Automating UI Generation by Model Composition.* In the 13th Conference on Automated Software Engineering (ASE'98), Oct. 13-16, 1998.

[33] S. Wilson, and P. Johnson. *Empowering Users in a Task-Based Approach to Design*, in Proceedings DIS'95, Symposium on Designing Interactive Systems, pages 25-31, ACM Press, 1995.

[34] S. Wilson, P. Johnson, et. al. *Beyond Hacking: a Model Based Approach to User Interface Design*, in People and Computers VIII, Proceedings of the HCI'93 Conference, Cambridge University Press.

[35] Spencer Rugaber and Richard Clayton. *The Representation Problem in Reverse Engineering.* In Proceedings of the First Working Conference on Reverse Engineering, Baltimore, Maryland. May 21-23, 1993.
http://www.cc.gatech.edu/reverse/repository/repr.ps

[36] Spencer Rugaber. *MASTERMIND Project Final Report.*
http://www.cc.gatech.edu/gvu/user_interfaces/Mastermind/final.html.

[37] Yannis Labrou and Tim Finin. *A Proposal for a new KQML Specification.* TR CS-97-03, February 1997, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250.

[38] Yigal Arens, Craig A. Knoblock and Chun-Nan Hsu. *Query Processing in the SIMS Information Mediator.* Advanced Planning Technology, editor, Austin Tate, AAAI Press, Menlo Park, CA, 1996.

[39] Zheng-Yang Liu, Mike Ballantyne and Lee Seward. *An Assistant for Re-Engineering Legacy Systems.*
http://www.spo.eds.com/edsr/papers/asstreeng.html.